Universidad Nacional del Sur

Tesina de Grado en Ciencias de la Computación

Titulo de la Tesina:
Subtitulo de la Tesina
Palabra suelta del subtitulo

Iñaki Garay

Bahía Blanca Argentina

2013

Iñaki Garay
igarai@gmail.com
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Bahía Blanca, Marzo 2013.

Índice general

Capítulo 1

Introducción

If your thesis is utterly vacuous
Use first-order predicate calculus
With sufficient formality
The sheerest banality
Will be hailed by the critics:
"Miraculous"

- Henry Kautz

1.1. Enunciado

El objetivo de este trabajo es presentar en detalle el diseño y la implementación del sistema de comunicación de percepciones entre agentes utilizado en el sistema multi-agente llamado `d3lp0r` desarrollado en el marco de MAPC 2011[1], la representación de la base de conocimiento utilizada, las decisiones de diseño e implementación tomadas, aspectos exitosos y las posibilidades de mejora.

En esta instancia de la competencia se hizo énfasis en lograr un Sistema Multi-Agente puramente distribuido en el cual cada agente realiza un proceso de toma de decisiones independiente en lugar de contar con una inteligencia centralizada que decide cuál será la acción a realizar para cada uno de los agentes.

Para asegurar que las decisiones de cada agente sean lo mas informadas posibles, se desarrolló un sistema de sincronización de la base de conocimiento que opera en una fase previa al proceso deliberativo de cada agente, asegurando que todos los agentes sepan lo mismo sobre el estado actual del mundo.

El objetivo particular de este proyecto es la aplicación de Argumentación para la implementación de diálogos entre agentes inmersos en un escenario con objetivos determinados. Puntualmente, se enfocará la investigación a la plataforma propuesta en el Multi-Agent Programming Contest, un juego académico donde agentes independientes compiten por diferentes objetivos. Sin embargo, el desarrollo de herramientas para implementar tales formalismos se encuentra en progreso y a un paso más lento.

[1] Multi-Agent Programming Contest 2011 - http://www.multiagentcontest.org/

Además, muchas de las herramientas disponibles carecen de una base formal y suelen ser simplemente un entorno de desarrollo amigable.

1.2. Organización del trabajo

El capítulo 2 da un conjunto de definiciones y un marco teórico sobre el cual se trabajó, abarcando los conceptos de agentes, arquitectura BDI, programacion lógica rebatible, y el contexto de desarrollo

El capítulo **??** describe la arquitectura general del sistema, la interna de cada agente, incluyendo la entrada y salida de datos, las fases de preprocesamiento de las percepciones, el esquema de representacion de conocimiento utilizada y el proceso deliberativo realizado por cada agente.

El capítulo **??** describe en detalle el servidor de percepciones: un sistema de sincronización de la base de conocimientos de cada agente, su funcionamiento interno, capacidades y limitaciones.

Capítulo 2

Definiciones preliminares

NOTA
La mayor parte de esta seccion es igual a la seccion correspondiente de la tesina de Diego Marcovecchio.

En este capítulo se revisarán algunas definiciones de conceptos técnicos, para posteriormente utilizarlos sin ambigüedad durante el resto de la presentación.

2.1. Agente inteligente

Un agente es una entidad computacional autónoma, que puede percibir su entorno a través de sensores, y actuar en dicho entorno utilizando efectores. Usualmente, la información que un agente percibe de su entorno es sólo parcial. Los agentes toman decisiones a partir de la información contenida en su base de conocimiento, siguiendo diferentes conjuntos de reglas propuestas, y actúan de manera acorde a la decisión tomada. Dichas acciones, a su vez, pueden producir efectos en el entorno.

Actualmente los agentes tienen un campo de aplicación muy amplio y existen muchos tipos de agentes diferentes (por ejemplo: *reactivos*, *deliberativos*, *inteligentes*, *de interface*, *colaborativos*), los cuales a su vez están orientados a distintos entornos de aplicación.

En la mayoría de los casos, los agentes no existen por sí solos, sino que participan de un Sistema Multi-Agente (SMA).

2.2. Sistema Multi-Agente

En un Sistema Multi-Agente (SMA) mas de un agente interactúan para lograr un objetivo o realizar una tarea común. Cada agente tiene información incompleta y capacidades limitadas, el control del sistema es distribuido, los datos están descentralizados, y la computación es asincrónica. Los agentes se

desenvuelven en un entorno dinámico y cambiante, el cual no puede predecirse y se ve afectado por las acciones que son llevadas a cabo.

Un aspecto importante en SMA es la comunicación entre agentes, la cual puede ser necesaria para que los agentes compitan o cooperen de acuerdo a sus metas individuales. Los diálogos con otros agentes del mismo ambiente son, actualmente, un área de estudio intensivo.

2.3. Modelo BDI

El *modelo Creencia-Deseo-Intención*, en adelante *BDI* (*Belief-Desire-Intention*), es un modelo desarrollado para el diseño de agentes inteligentes, basado en una vista simplificada de la inteligencia humana. Como se analizará en la sección **??**, el sistema presentado en este trabajo implementa una adaptación de dicho modelo. Por esta razón, se introducen en esta sección los conceptos básicos relacionados, que sirvieron de base para nuestro desarrollo.

El modelo BDI está dedicado al modelado formal del razonamiento práctico, es decir, la formalización de las bases y explicaciones psicológicas y filosóficas (provenientes principalmente de la filosofía de la mente y de la acción) de los conceptos de agente, acción, intención, creencia, voluntad, deliberación, razonamiento de medios y fines, etc. El razonamiento práctico es incorporado en agentes (por ejemplo, los seres humanos) capaces de perseguir y, por lo tanto, comprometerse con una determinada meta factible (una acción en particular) a través de una cuidadosa planificación de los medios, de las condiciones previas y las acciones que conducen a ese objetivo.

Estos conceptos son incorporados al modelo mediante la implementación de los aspectos principales de la teoría del razonamiento práctico humano de Michael Bratman (también referido como Belief-Desire-Intention, o BDI). Es decir, implementa las nociones de creencia, deseo y (en particular) intención, de una manera inspirada por Bratman. Una discusión más extensa puede ser encontrada en el mencionado trabajo de Bratman[**?**] y Searle[**?**].

Este basamento teórico permite al modelo resolver un problema particular que se presenta en la programación de agentes. Provee un mecanismo para separar la actividad de seleccionar un plan de la ejecución de los planes actualmente activos. Los agentes BDI son capaces de balancear el tiempo invertido en deliberar sobre los planes (elegir qué hacer) y ejecutar estos planes (llevarlo a cabo). La actividad de crear los planes en primera instancia, escapa al alcance del modelo.

2.3.1. Creencias, Deseos e Intenciones

Las *creencias, deseos e intenciones* son consideradas estados mentales intencionales (de forma opuesta a, por ejemplo, el dolor o el placer). Las *creencias* describen la percepción de la realidad a través de datos provenientes de los sentidos. Representan el estado *informacional* del agente; comprenden el conocimiento (tanto de sentido común como teórico) sobre el mundo, ya sea externo o interno. Están sujetas a revisión, lo que implica que pueden cambiar en el futuro, pueden ser rechazadas o agregadas.

Los *deseos* e *intenciones*, pueden ser vistos como conceptos que se asemejan, aunque con algunas sutiles diferencias. Los deseos representan el estado *motivacional* del agente; consisten en su voluntad de alcanzar ciertos objetivos o situaciones. Entre los deseos, se distingue la noción de *meta*. Una meta es un deseo que ha sido adoptado por el agente para ser perseguido activamente. Esta definición impone la restricción de que el conjunto de metas, o deseos activos, debe ser consistente.

Por último, el concepto de intención representa el estado *deliberativo* del agente, lo que el agente ha elegido hacer. Constituyen deseos para los cuales el agente se ha comprometido. Es una noción más ligado al compromiso que es asumido, en función alcanzar los estados o situaciones deseadas.

2.3.2. Deliberación y planificación

Por *deliberación* entendemos lo que la literatura denomina *silogismo práctico*, es decir, la inferencia de una intención a partir de un conjunto de creencias y deseos. Esto es, la selección de un deseo factible. Una *decisión* consiste en el último paso de este proceso de inferencia mediante el cual resulta electo uno de muchos deseos y potenciales intenciones. Es, por esto, un concepto ligado directamente al de intención. Definir una intención implica, en términos de agentes implementados, comenzar la ejecución de un *plan*.

Una *acción* puede ser definida, intuitivamente, como la ejecución de una operación que causa un determinado efecto o consecuencia sobre el entorno en el cual se está desempeñando el agente. La *planificación* consiste en la disposición de una secuencia de acciones con el fin de lograr una (o más) de sus intenciones de alcanzar una meta. Los planes pueden ser complejos en mayor o menor medida, en función a la cantidad de acciones que contiene. En particular, los planes pueden contener otros planes, dado que satisfacer una meta puede requerir la satisfacción de metas intermedias. Esto refleja que en el modelo de Bratman, inicialmente los planes son concebidos sólo parcialmente, y los detalles son incorporados a medida que progresa su ejecución.

2.4. Programación Lógica Rebatible

A continuación introducimos las definiciones básicas necesarias para representar conocimiento en Programación Lógica Rebatible (DeLP). Para un tratamiento exhaustivo, se remite al lector interesado al trabajo de A. García y G. Simari[**?**]. En lo que sigue, se asume que el lector posee un conocimiento básico acerca de los aspectos fundamentales de la programación lógica.

Definición 2.4.1. (Programa DeLP $\mathcal{P}$)

Un programa lógico rebatible (delp) es un conjunto $\mathcal{P} = (\Pi, \Delta)$ donde Π y Δ representan conjuntos de conocimiento *estricto* y *rebatible*, respectivamente. El conjunto Π de conocimiento estricto involucra *reglas estrictas* de la forma $L \leftarrow Q_1, \ldots, Q_k$ y *hechos* (reglas estrictas con cuerpo vacío), y se asume que es *no-contradictorio*. El conjunto Δ de conocimiento rebatible involucra *reglas rebatibles* de la forma $L -\!\prec Q_1, \ldots, Q_k$, lo cual se interpreta como "$Q_1, \ldots, Q_k$ proveen razones tentativas para creer L". Las reglas estrictas y rebatibles en DeLP son definidas usando un conjunto finito de literales. Un literal es un átomo (L), la negación estricta de un átomo ($\sim L$) o la negación *default* de un átomo ($not\ L$).

El lenguaje lógico subyacente en DeLP es el de la programación lógica extendida, enriquecido con el símbolo especial " $-\!\prec$ " para denotar reglas rebatibles. Tanto la negación *default* como la clásica están permitidas (denotadas *not* y $\sim$, respectivamente). Sintácticamente, el símbolo " $-\!\prec$ " es lo único que distingue un regla *rebatible* $L -\!\prec Q_1, \ldots, Q_k$ de una regla *estricta* (no- rebatible) $L \leftarrow Q_1, \ldots, Q_k$. Las reglas DeLP por lo tanto, son consideradas como *reglas de inferencia* en lugar implicaciones. De forma análoga a la programación lógica tradicional, la *definición* de un predicado P en $\mathcal{P}$, denotado $P^{\mathcal{P}}$, está dada por el conjunto de todas las reglas (estrictas y rebatibles) con cabeza P y aridad n en $\mathcal{P}$. Si P es un predicado en $\mathcal{P}$, entonces *nombre(P)* y *aridad(P)* denotan el nombre y la aridad del predicado, respectivamente. Escribiremos $\mathsf{Pred}(\mathcal{P})$ para denotar el conjunto de todos los nombres de predicados definidos en un programa DeLP $\mathcal{P}$.

2.4.1. Argumento, Contraargumento y Derrota

Dado un programa DeLP $\mathcal{P} = (\Pi, \Delta)$ resolver consultas resulta en la construcción de *argumentos*. Un argumento $\mathcal{A}$ es un conjunto (posiblemente vacío) de reglas rebatibles fijas que junto al conjunto Π provee una prueba lógica para un dado literal $\mathcal{Q}$, satisfaciendo los requerimientos adicionales de *no-contradicción* y *minimalidad*. Formalmente:

Definición 2.4.2 (Argumento)**.** Dado un programa DeLP $\mathcal{P}$, un argumento $\mathcal{A}$ para una consulta $\mathcal{Q}$, notado $\langle \mathcal{A}, \mathcal{Q} \rangle$ es un subconjunto de instancias fijas de las reglas rebatibles en $\mathcal{P}$, tal que:

(1) existe una derivación rebatible para $\mathcal{Q}$de $\Pi \cup \mathcal{A}$;

(2) $\Pi \cup \mathcal{A}$ es no-contradictorio (i. e., $\Pi \cup \mathcal{A}$ no implica dos literales complementarios L y $\sim L$ (o L y not L), y,

(3) $\mathcal{A}$ es minimal con respecto al conjunto inclusión (i. e., no hay $\mathcal{A}' \subset \mathcal{A}$ tal que existe una derivación rebatible para $\mathcal{Q}$ de $\Pi \cup \mathcal{A}'$).

Un argumento $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ es un *subargumento* de otro argumento $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$ si $\mathcal{A}_1 \subseteq \mathcal{A}_2$. Dado un programa DeLP $\mathcal{P}$, *Args(P)* denota el conjunto de todos los posibles argumentos que pueden ser derivados de $\mathcal{P}$.

La noción de derivación rebatible corresponde a la usual derivación SLD dirigida por consultas empleada en programación lógica, aplicando *backward chaining* a las reglas estrictas y rebatibles; en este contexto, un literal negado $\sim P$ es tratado simplemente como un nuevo nombre de predicado *no_P*. La minimalidad impone una especie de "principio de la navaja de Occam" sobre la construcción de argumentos. El requerimiento de no-contradicción prohíbe el uso de (instancias fijas de) reglas rebatibles en un argumento $\mathcal{A}$ cuando $\Pi \cup \mathcal{A}$ deriva dos literales complementarios. Es de notar que el concepto de no-contradicción captura los dos enfoques usuales de negación en la programación lógica (negación *default* y negación clásica), ambas presentes en DeLP y relacionadas a la noción de contraargumento, como se muestra a continuación.

Definición 2.4.3 (**Contraargumento**). Un argumento $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ es un *contraargumento* para un argumento $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$ si y sólo si

a) (ataque a subargumento) existe un subargumento $\langle \mathcal{A}, \mathcal{Q} \rangle$ de $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$ (llamado *subargumento en desacuerdo*) tal que el conjunto $\Pi \cup \{\mathcal{Q}_1, \mathcal{Q}\}$ es contradictorio, o

b) (ataque por negación default) un literal $\mathsf{not}\mathcal{Q}_1$ está presente en el cuerpo de alguna regla en $\mathcal{A}_2$.

La primer noción de ataque es tomada del framework de Simari-Loui; la última está relacionada al enfoque argumentativo de programación lógica de Dung, asi como también a otras formalizaciones, como el trabajo de Prakken y Sartor, o el trabajo de Kowa y Toni.

Como en muchos marcos de argumentación, vamos a asumir un *criterio de preferencia* para los argumentos en conflicto definido como la relación $\preceq$, la cual es un subconjunto del producto cartesiano *Args(*$\mathcal{P}$*)* $\times$ *Args(*$\mathcal{P}$*)*. Esto lleva a la noción de *derrota* entre argumentos como una refinación del criterio de contraargumento. En particular, vamos a distinguir entre dos tipos de derrotadores, *propios* y *por bloqueo*.

Definición 2.4.4 (**Derrotadores propios y por bloqueo**). Un argumento $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ es un *derrotador propio* para un argumento $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$ si $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ contra-argumenta $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$ con un sub-argumento en desacuerdo $\langle \mathcal{A}, \mathcal{Q} \rangle$ (ataque a subargumento) y $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ es estrictamente preferido sobre $\langle \mathcal{A}, \mathcal{Q} \rangle$ con respecto a $\preceq$.

Un argumento $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ es un *derrotador por bloqueo* para un argumento $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$ si $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ contra-argumenta $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$ y una de las siguientes situaciones se presenta: (a) Hay un sub-argumento en desacuerdo $\langle \mathcal{A}, \mathcal{Q} \rangle$ para $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$, y $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ y $\langle \mathcal{A}, \mathcal{Q} \rangle$ no están relacionados entre sí con respecto a $\preceq$; o (b) $\langle \mathcal{A}_1, \mathcal{Q}_1 \rangle$ es un ataque por negación default sobre algún literal $\mathsf{not}\mathcal{Q}_1$ en $\langle \mathcal{A}_2, \mathcal{Q}_2 \rangle$.

El término *derrotador* será usado para referirse indistintamente a derrotadores propios o por bloqueo.

La especificidad generalizada es típicamente usada como un criterio de preferencia basado en la sintaxis para argumentos en conflicto, favoreciendo aquellos argumentos que están *más informados* o son *más directos*. A modo de ejemplo, consideremos tres argumentos

$$\langle \{a \multimap b, c\}, a \rangle,\ \langle \{\sim\!\mathrm{a} \multimap b\}, \sim a \rangle \text{ y } \langle \{(a \multimap b); (b \multimap c)\}, a \rangle$$

construidos sobre la base de un programa

$$\mathcal{P} = (\Pi, \Delta) = (\{b, c\}, \{b \multimap c; a \multimap b; a \multimap b, c; \sim a \multimap b\}).$$

Si se utiliza especificidad generalizada como criterio de comparación entre argumentos, el argumento $\langle \{a \multimap b, c\}, a \rangle$ sería preferido sobre el argumento $\langle \{\sim\!\mathrm{a} \multimap b\}, \sim a \rangle$ ya que el primero es considerado *más informado* (i. e., está basado en más premisas). Sin embargo, el argumento $\langle \{\sim\!\mathrm{a} \multimap b\}, \sim a \rangle$ es preferido sobre $\langle \{(a \multimap b); (b \multimap c)\}, a \rangle$ ya que el primero es considerado *más directo* (i. e., es obtenido a partir

de una derivación más corta). Sin embargo, debe ser remarcado que, además de especificidad, otros criterios de preferencia alternativos pueden ser usados; e. g., aplicar prioridad sobre las reglas para definir la comparación de argumentos, o considerar valores numéricos correspondientes a medidas asociadas a conclusiones de argumentos. El primer enfoque es empleado en D-PROLOG, Lógica Rebatible, extensiones de la Lógica Rebatible, y programación lógica sin negación por falla. El segundo criterio fue el aplicado en el desarrollo del sistema que se presenta en los capítulos siguientes.

2.4.2. Cómputo de garantías a través de análisis dialéctico

Dado un argumento $\langle \mathcal{A}, \mathcal{Q} \rangle$, pueden existir diferentes derrotadores $\langle \mathcal{B}_1, \mathcal{Q}_1 \rangle \ldots \langle \mathcal{B}_k, \mathcal{Q}_k \rangle$, k ≥ 0 para $\langle \mathcal{A}, \mathcal{Q} \rangle$. Si el argumento $\langle \mathcal{A}, \mathcal{Q} \rangle$ es derrotado, entonces ya no estaría soportando su conclusión $\mathcal{Q}$. Sin embargo, dado que los derrotadores son argumentos, estos pueden a su vez ser derrotados. Esto induce un análisis dialéctico recursivo completo para determinar qué argumentos son derrotados en última instancia. Para caracterizar este proceso, primero se introducen algunas nociones auxiliares.

Una *línea argumentativa* comenzando en un argumento $\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle$ (denotado $\lambda^{\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle}$) es una secuencia $[\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle, \langle \mathcal{A}_1, \mathcal{Q}_1 \rangle, \langle \mathcal{A}_2, \mathcal{Q}_2 \rangle, \ldots, \langle \mathcal{A}_n, \mathcal{Q}_n \rangle \ldots]$ que puede ser pensada como un intercambio de argumentos entre dos partes, un *proponente* (argumentos en posiciones pares) y un *oponente* (argumentos en posiciones impares). Cada $\langle \mathcal{A}_i, \mathcal{Q}_i \rangle$ es un derrotador para el argumento previo $\langle \mathcal{A}_{i-1}, \mathcal{Q}_{i-1} \rangle$ en la secuencia, $i > 0$.

A fin de evitar razonamiento *falaz* o mal-formado (e. g., lineas argumentativas infinitas), el análisis dialéctico impone restricciones adicionales para que el intercambio de argumentos pueda ser considerado racionalmente *aceptable*. Puede ser probado que las líneas argumentativas aceptables son finitas. Un tratamiento exhaustivo sobre restricciones de aceptabilidad pueden ser encontradas en el trabajo de García y Simari[**?**].

Dado un programa DeLP $\mathcal{P}$ y un argumento inicial $\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle$, el conjunto de todas las líneas argumentativas aceptables comenzando en $\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle$ da lugar a un análisis dialéctico completo para $\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle$ (i. e., todos los diálogos posibles sobre $\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle$ entre proponente y oponente), formalizado mediante un *árbol dialéctico*.

Los nodos en un árbol dialéctico $T_{\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle}$ pueden ser marcados como nodos *derrotados* y *no derrotados* (nodos D -*defeated*- y nodos U -*undefeated*-, respectivamente). Un árbol dialéctico será marcado como un árbol AND-OR: todas las hojas en $T_{\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle}$ serán marcadas como nodos U (dado que no poseen derrotadores), y cada nodo interno será marcado como nodo D si y sólo si tiene al menos un nodo U como hijo, y como nodo U en otro caso. Un argumento $\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle$ es finalmente aceptado como válido (o *garantizado*) con respecto a un programa DeLP $\mathcal{P}$ si y sólo si la raíz del árbol dialéctico asociado $T_{\langle \mathcal{A}_0, \mathcal{Q}_0 \rangle}$ está etiquetado como *nodo U*.

Dado un programa DeLP $\mathcal{P}$, resolver una consulta $\mathcal{Q}$ con respecto a $\mathcal{P}$ implica determinar si $\mathcal{Q}$ está soportado por (al menos) un argumento garantizado. Diferentes actitudes doxásticas distinguidas de la siguiente manera:

(1) textitYes: se cree ArgQ si y sólo si hay al menos un argumento garantizado soportando ArgQ en base a PP.

(2) *No*: se cree $\sim\mathcal{Q}$ si y sólo si hay al menos un argumento garantizado soportando $\sim\mathcal{Q}$ en base a $\mathcal{P}$.

(3) textitUndecided: ni ArgQ ni litno ArgQ están garantizados con respecto a PP.

(4) textitUnknown: ArgQ no se encuentra en el lenguaje de PP.

2.5. Multi-Agent Programming Contest

El *Multi-Agent Programming Contest* es un concurso de programación de Inteligencia Artificial iniciado en el año 2005 con el objetivo de estimular la investigación en el área de desarrollo y programación de Sistemas Multi-Agente. Para ello, la competencia propone diferentes escenarios de juego de manera anual, que obligan a los participantes tanto a identificar y resolver problemas clave, como a explorar lenguajes, plataformas y herramientas de programación para Sistemas Multi-Agente.

2.5.1. Escenario MAPC 2011

El escenario del año 2011 está formado por el mapa de un planeta representado mediante un grafo. Cada nodo del grafo es una locación válida (y tiene un valor determinado), y existen arcos (con diferente costo de energía) que permiten a un agente desplazarse de una locación a otra.

En cada ronda de la competición participan dos equipos rivales. Cada equipo posee un conjunto de agentes con diferentes roles preestablecidos (*Explorador*, *Saboteador*, *Reparador*, *Sentinela* e *Inspector*). El rol de cada agente define tanto el conjunto de acciones que puede realizar, como sus características físicas (*Energía*, *Salud*, *Fuerza* y *Rango de Visión*).

Puntaje

La simulación del juego se desarrolla por pasos, y en cada paso se otorga a los equipos una determinada cantidad de puntos según el estado de la simulación. El objetivo del juego es obtener la mayor cantidad de puntos posibles cuando la simulación termina.

Para obtener puntos, los agentes de cada uno de los equipos deben lograr formar «*zonas*» en el mapa logrando posicionarse en diferentes locaciones de manera estratégica.

La predominancia de un equipo sobre el otro en los nodos es determinada por un algoritmo bien definido para la competencia, y el valor de todos los nodos dominados por un equipo es el principal factor del puntaje otorgado en cada uno de los pasos de la simulación.

Algunas otras situaciones, como el logro de determinados *achievements*, pueden otorgar puntos adicionales y dinero al equipo.

Acciones

Todos los agentes tienen acciones en común que pueden realizar en cada uno de los pasos de la simulación:

- goto(X): el agente se desplaza hacia el nodo X, siempre y cuando exista un arco que conecte el nodo actual del agente con X, y dicho arco tenga un costo menor a la energía actual del agente.
- survey(X): el agente recibe en su próxima percepción los costos de todos los arcos conectados al nodo en el que se encuentra actualmente.
- buy(X): el agente utiliza el dinero obtenido a partir de los *achievements* para aumentar el valor máximo de cualquiera de sus características físicas (Energía, Salud, Fuerza o Rango de visión) en 1 punto.
- recharge: el agente recupera el 20 % de su energía máxima.
- skip: el agente pasa al turno siguiente sin realizar ningún tipo de acción.

Además, según el rol de cada agente, existen algunas acciones específicas que pueden realizar:

- attack(X): acción disponible únicamente para los *Saboteadores*; el agente ataca a un enemigo X, si dicho enemigo se encuentra en el mismo nodo. El ataque, de tener éxito, decrementa la energía del agente enemigo, pudiendo deshabilitarlo en caso de que ésta llegue a 0.
- parry: acción disponible únicamente para los *Reparadores*, textitSaboteadores y textitSentinelas. La acción protege al agente de los ataques enemigos, impidiendo que éstos tengan éxito.
- probe: acción disponible únicamente para los *Exploradores*. El agente recibe en su próxima percepción el valor del nodo en el que se encuentra actualmente. Ésta acción no sólo resulta importante por conocer el valor del nodo, sino que además permite que, cuando el nodo es conquistado por el equipo, dicho valor se sume al total de puntos de la zona. Un nodo en el que no se realizó *probe* suma únicamente 1 punto al valor total de la zona.
- inspect: acción disponible únicamente para los *Inspectores*. El inspector recibe en su próxima percepción la información física (Salud, Energía, Fuerza, Rango de visión) de todos los agentes enemigos que se encuentren en el mismo nodo que él, o en cualquier vecino directo.
- repair(X): acción disponible únicamente para los *Reparadores*. El reparador aumenta el valor de la Salud actual de su compañero de equipo X (volviendo a habilitarlo, en caso de que su Salud fuera 0).

Capítulo 3

Arquitectura

DRAFT

Este capítulo describe la arquitectura general del sistema, el protocolo de comunicación con el servidor MASSIM, la arquitectura interna de cada agente, incluyendo la entrada y salida de datos, las fases de preprocesamiento de las percepciones, el esquema de representacion de conocimiento utilizada y el proceso deliberativo realizado por cada agente.

3.1. Arquitectura del sistema

El sistema `d3lp0r` consiste del conjunto de procesos Agentes y el proceso Servidor de Percepciones. Los Agentes tienen como componentes principales al módulo principal, que dirije la lógica de control del agente, los módulos de comunicación con el servidor MASSim y el Servidor de Percepciones, y los módulos de establecimiento de creencias y deliberacion. El Servidor de Percepciones esta implementado en un único módulo, y tiene como componentes principales la lógica de control y el protocolo de comunicación.

El programa principal del agente esta implementado en Python y maneja toda comunicación con los servidores, parseo de los mensajes XML, procesamiento de la información contenida en las percepciones para transformarlas a un formato adecuado para su aserción en la base de conocimiento del agente, y la generación del XML que representa las acciones que toma el agente y es enviado al servidor MASSim.

La inicialización del agente consiste en la apertura de la conexión al servidor MASSIm y la subsiguiente autentificación, la apertura de la conexión al Servidor de Percepciones, y la inicialización del motor Prolog. Luego de la fase de inicialización se ingresa al bucle principal, en el cual se reciben y parsean mensajes del servidor MASSim y se responde a ellos de manera adecuada.

Al recibir un mensaje de tipo `sim-start`, la información presente en el mensaje tal como el rol del agente y los parametros de la simulación se asertar en la base de conocimiento del agente y se inicia el ciclo de percibir-actuar.

Cada iteracion del bucle de percibir-actuar espera un mensaje de tipo `request-action` desde el servidor MASSim y parsea el XML para transformarlo en un diccionarion (arreglo asociativo) de Python. Los elementos de la percepción se separan en una sección "publica", la cual es enviada al Servidor de Perceciones, y otra "privada".

A continuación el agente enviará la sección pública de su percepción al Servidor de Percepciones y espererá la "percepción global" que contendrá el resto de la información percibida por el equipo. La percepción global se unirá con su propia, y será asertada en su b ase de conocimiento, estableciendo sus creencias.

El módulo de decisión implementado en Prolog es consultado para determinar la siguiente acción que ejecutará el agente. Una vez que el flujo de control retorna al programa Python con el resultado de la fase deliberativa, el mensaje XML correspondiente es generado y enviado al servidor MASSim.

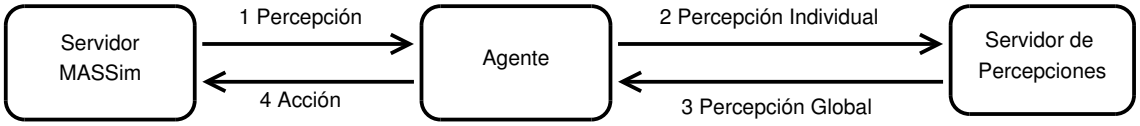

Figura 3.1: Diagrama de la secuencia de acciones de cada ciclo de la simulación.

3.2. Conexión del Agente con el Servidor MASSim

El protocolo de comunicación con el servidor MASSim especificado por el enunciado del concurso MAPC define que los agentes y el servidor intercambian mensajes en formato XML codificados por UTF-8, con un byte nulo para indicar el final del mensaje.

3.2.1. Protocolo de comunicación con el servidor MASSim

TODO

3.2.2. Formato de mensajes

TODO

Percepciones

TODO

Acciones

Las acciones se representan con la cadena de caracteres:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <message type="action">
    <action id=\"ID\" type=\"TYPE\"/>
  </message>\0
```

En el caso de las acciones que requieren un parámetro adicional, la representación es:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <message type="action">
    <action id="ID" param="PARAM" type="TYPE"/>
  </message>\0
```

donde `ID` es el identificador de mensaje enviado por el servidor MASSim en la percepcion, `TYPE` es el tipo de accion y `PARAM` es el parametro.

`TYPE` puede ser uno de:

- `skip`
- `goto`
- `attack`
- `parry`
- `probe`
- `survey`
- `inspect`
- `repair`
- `recharge`
- `buy`

TODO

Capítulo 4

Servidor de Percepciones

The PS maintains a connection for each agent. The connection handling methods encode the associate array into a form suitable for conversion into a set datastructure, which is then sent over the network. On each iteration, the PS waits for each agent's data, performs a union of all data sets, and returns to each agent the set difference between the data the agent sent and the total union. Figures **??** and **??** show example percepts after parsing, before being sent to the percept server.

```
{ 'surveyed_edges' : [ ],
  'vis_verts'      : [ { 'name': 'vertex65',
                         'team': 'none' },
                       ... ],
  'vis_ents'       : [ { 'node':   'vertex97',
                         'status': 'normal',
                         'name':   'a6',
                         'team':   'A' },
                         ... ],
  'inspected_ents' : [ ],
  'vis_edges'      : [ { 'node1': 'vertex141',
                         'node2': 'vertex65' },
                       ... ],
  'position'       : [ { 'node':       'vertex141',
                         'vis_range':  '1',
                         'health':     '6',
                         'name':       'self',
                         'max_health': '6' } ],
  'probed_verts'   : [ ] }
```

Figura 4.1: A sample public section of a percept, after parsing.

```
{ 'total_time':      2000L, 'zone_score': '0',
  'last_step_score': '20',  'timestamp':  '1323732915832',
  'strength':        '0',   'energy':     '11',
  'money':           '12',  'max_energy_disabled': '16',
  'last_action':     'recharge', 'max_health':     '6',
  ... }
```

Figura 4.2: A sample private section of a percept.

4.1. Conexión del Agente con el Servidor de Percepciones

TODO

4.2. Formato de mensajes

TODO

4.3. Protocolo de Comunicación con el Servidor de Percepciones

TODO

4.3.1. Esquema de Codificación

Tras la recepción de la percepción del servidor, el parseo del XML y su transformación en diccionarios nativos de Python, el agente retransmite la informacion recibida al servidor de percepciones. Aunque es factible enviar la percepcón en formato XML o como la representación de la estructura de datos en memoria del agente, unos de los objetivos es distribuir tanto como posible el trabajo realizado por el sistema. Como las tareas principales realizadas por el servidor de percepciones son la union de la información y la diferencia de cada percepción individual con el total, se implemento un esquema más de codificacion. Se transforma el diccionario que contiene la percepción en una estructura sobre la cual se pueden realizar operaciones de conjunto de manera eficiente.

TODO

4.3.2. Esquema de Reconexión para Agentes Caidos

Si el agente pierde la conexión,

4.3.3. Evaluación de Performance

Bibliografía

[1] F. Bellifemine, A. Poggi, G. Rimassa, *JADE - A FIPA-compliant agent framework*. CSELT internal technical report, 1999.

[2] F. Bellifemine, A. Poggi, G. Rimassa, *Developing multi-agent systems with a FIPA-compliant agent framework*. Software - Practice And Experience, 2001.

[3] R. S. Cost, Y. Labrou, T. Finin, *Coordinating Agents using Agent Communication Languages Conversations*. 2000.

[4] Dastani et al., *A Programming Language for Cognitive Agents: Goal Directed 3APL*. Proceedings of the First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools, 2003.

[5] Doran et al., *On Cooperation in Multi-Agent Systems. The Knowledge Engineering Review*. 1996.

[6] A. J. García, G. R. Simari, *Defeasible logic programming: An argumentative approach*. Journal of Theory and Practice of Logic Programming, 2004.

[7] A. J. García, M. Tucat, G. R. Simari, *Interaction Primitives for Implementing Multi-agent Systems*. Interaction Primitives for Implementing Multi-agent Systems, 2005.

[8] M. Huhns, L. Stephens, *Multiagent Systems and Societies of Agents*. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, 1999.

[9] N. R. Jennings et al., *Argumentation-Based Negotiation*. Proceedings of the International Workshop on Multi-Agent Systems, 1998.

[10] S. Kalenka, N. R. Jennings, *Socially Responsible Decision Making by Autonomous Agents*. Proceedings of the 5th International Colloquium on Cognitive Science, 1999.

[11] J. Müller, *Negotiation Principles*. Foundations of Distributed Artificial Intelligence, 1996.

[12] A. Rao, M. Georgeff, *Modeling rational agents within a BDI-architecture*. Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, 1991.

[13] A. S. Rao, M. P. Georgeff, *BDI Agents: From Theory to Practice*. Proceedings of the First International Conference on Multi-Agent Systems, 1995.

[14] S. V. Rueda, A. J. García, G. R. Simari, *Argument-based Negotiation among BDI Agents*. Journal of Computer Science and Technology, 2002.

[15] M. Wooldridge, *Intelligent Agents*. Multiagent Systems, The MIT Press, 1999.

[16] M. E. Bratman, *Intention, Plans, and Practical Reason*. Cambridge University Press, 1999.

[17] J.R. Searle, *Intentionality, an Essay in the Philosophy of Mind*. Cambridge University Press, 1983.

Bibliografía

[Gar00] García, A. J. *Programación en Lógica Rebatible: Lenguaje, Semántica Operacional y Paralelsimo*. PhD thesis, Departemento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, December 2000.

Esta tesis define el lenguaje de la Programación en Lógica Rebatible (DeLP) y especifica su semántica operacional. El procedimiento de prueba de DeLP consiste de un análisis dialéctico el cual considera argumentos a favor o en contra de la consulta efectuada. Se presenta un modelo de computo secuencial y otro en paralelo. Se desarrollan aplicaciones para el lenguaje definido.