T1 – Perspectiva conceptual.

La IA tiene dos concepciones claramente diferenciadas, las cuales tienen también sus propios objetivos, métodos y enfoques diferentes. Por ello cabe distinguir entre:

1.1. La IA como Ciencia

La IA entendida como ciencia es básicamente una tarea de análisis. Su ámbito de conocimiento engloba el conjunto de hechos asociados a la neurología y la cognición, como pueden ser la percepción, la memoria, el lenguaje, etc.

La perspectiva científica busca el desarrollo de una teoría computable del conocimiento humano, es decir, una teoría que pueda ejecutarse en un sistema de cálculo con fines predictivos (construir programas que emulen el comportamiento inteligente de los humanos). Se ve, por tanto, el ambicioso objetivo perseguido por este enfoque científico de la IA, y el porqué se llama *hipótesis fuerte* a esta idea.

Existe un enfoque conocido como *hipótesis débil*, que persigue un objetivo más modesto, como es el desarrollar máquinas que exhiban un comportamiento inteligente, no necesariamente emulando el pensamiento humano.

1.2. La IA como Ingeniería

La IA como ingeniería también tiene sus dificultades. La primera, es que el objeto formal de la IC (Ingeniería del Conocimiento) es el propio conocimiento, y éste es pura forma. La idea se basa en la estructura relacional y en un punto común para los distintos observadores, que dotan de significado a los símbolos formales y físicos que constituyen el cálculo.

La segunda dificultad se basa en que todavía no se dispone de una sólida *teoría del conocimiento*, de la cual se debe encargar de desarrollar la parte de la IA como ciencia. Es decir, no tenemos los conocimientos sobre neurofisiología o procesos cognitivos suficientes para poder apoyarse en ellos desde la parte ingenieril.

En la mayoría de los desarrollos de la IC, llamados Sistemas Basados en Conocimiento (SBCs), y anteriormente conocidos como Sistemas Expertos (SEs) se procede en base a los mismos pasos:

- Se parte de la descripción en lenguaje natural del problema, descripción generalmente realizada por un experto (análisis).

- Después, se modela esta descripción mediante diferentes paradigmas (simbólico, conexionista, situado) y se llega a obtener un modelo conceptual (diseño).

- A partir de este modelo, y utilizando diversos operadores formales, como las distintas lógicas, las RNAs (redes neuronales artificiales), etc, se llega a un modelo formal, el cual ya es computable por un sistema de cálculo (implementación).

T2 – Aspectos metodológicos (paradigmas).

Se puede considerar un paradigma como una aproximación metodológica a la IA y a la IC que ha sido consensuada entre un grupo de profesionales del campo que la consideran como la forma normal de hacer ciencia o ingeniería.

Paradigma es sinónimo de *forma de abordar un problema*. Aplicado a la IC, paradigma es una forma de modelar, formalizar, programar e implementar físicamente el soporte de esos programas (por ejemplo, en el *cuerpo* de un robot).

De forma general, los paradigmas se pueden dividir entre los basados en representaciones y los basados en mecanismos, aunque aquí vamos a considerar 4 paradigmas distintos:

2.1. El paradigma simbólico

Este paradigma considera que todo el conocimiento necesario para resolver una tarea de diagnóstico, planificación, control o aprendizaje, puede representarse usando descripciones declarativas y explícitas en lenguaje natural. Estas descripciones declarativas estarían formadas por un conjunto de hechos, y otro conjunto de reglas de inferencia que describen las relaciones estáticas y dinámicas entre esos hechos.

En muchas ingenierías, es usual distinguir tres tipos de tareas para resolver los problemas específicos de cada una de ellas (análisis, síntesis y modificación/mantenimiento). Así, aplicando estos términos a la IC tenemos que en la fase de *análisis* disponemos de todas las soluciones posibles al problema, y el trabajo es básicamente de clasificación y elección de esas soluciones en base a la descripción del problema. En la fase de *síntesis* nos basamos principalmente en un trabajo de diseño y construcción con restricciones (los requisitos del problema). Finalmente, tenemos que la fase de *modificación* tiene que ver sobre todo con el ajuste de parámetros en las estructuras ya diseñadas y sintetizadas.

En este paradigma, las entidades del dominio (hechos o reglas) representan roles de entrada y/o salida para las distintas inferencias (reglas), las cuales generan el resultado final del *razonamiento*. Es decir, usamos nuestra *base de conocimiento* para evaluar y obtener nuevos hechos y reglas, que actualizan nuestro modelo del medio mediante aprendizaje, y pasan a formar parte también de la *base del conocimiento*.

Este paradigma es adecuado para aquellas aplicaciones en las que disponemos de conocimiento suficiente para especificar reglas inferenciales, y en procesos de aprendizaje inductivo en los que también disponemos de conocimiento suficiente para especificar las "meta-reglas" que actualizarán nuestra base del conocimiento.

2.2. El paradigma situado

También llamado reactivo, o basado en conductas, está basado en que toda percepción y toda acción están estructuralmente acopladas, mediante perceptores y efectores concretos, a un medio externo e interno también concretos.

El sistema que queremos modelar mediante este paradigma se encuentra en un medio que forma un lazo de realimentación mediante sus sensores y efectores. Así, todo lo que no pueda ser percibido por los sensores no existe para el sistema, y tampoco podrá ejecutar acciones que no puedan realizar sus propios efectores.

En este esquema, los roles de entrada vienen a ser las propias percepciones captadas por sus sensores, los roles de salida serán las acciones que ejecutarán sus efectores, y todo el motor de inferencia serán esquemas de asociación precalculados, o autómatas finitos, que permitirán que el sistema actúe de forma *reactiva*, sin tener que estar deliberando las posibles salidas. Además, las posibles acciones también estarán precalculadas.

La lógica interna depende de las coordinaciones espacio-temporales entre los estados actuales de los dos tipos de mecanismos descritos: los perceptuales y los motores.

Este paradigma se usa esencialmente en robótica y en aplicaciones en tiempo real simples. Es decir, en ámbitos donde la interfaz del sistema informático no es humana, sino que intercambia datos con el medio mediante sus sensores y efectores. Cuando aumenta la complejidad del sistema, se hace necesario utilizar soluciones híbridas, combinando componentes reactivas (rápidas) con otras deliberativas (lentas).

2.3. El paradigma conexionista

En este paradigma, la representación del conocimiento se realiza mediante el uso de líneas numéricas etiquetadas para la entrada y salida de una RNA (red neuronal artificial), y la inferencia se resuelve mediante un clasificador numérico parametrizado en el que el valor de los parámetros se ajusta mediante aprendizaje.

Un agente conexionista tiene una estructura modular, con un gran número de procesadores elementales (*neuronas*) interconectados, los cuales evalúan una función de cálculo local. Sus características distintivas son:

- Todos los problemas resueltos con RNAs se resuelven como un clasificador numérico adaptativo, que asocia valores de entrada de un conjunto de *observables* con valores de salida de otro conjunto más reducido de *clases*.

- Mucho del conocimiento disponible se obtiene de una fase de *análisis de los datos*, en los que es el observador externo quien decide cuales van a ser las variables de entrada y salida, el tipo de cálculo local, la estructura interna en capas de la RNA, etc.

- Hay que tener en cuenta el balance entre datos y conocimiento disponible. Si los datos son etiquetados (se conoce la respuesta de la red) se usan en aprendizaje supervisado y en una fase final de validación de la red. Mientras que si son datos no etiquetados, se usan en aprendizaje autoorganizativo (no supervisado), para un preproceso de los mismos.

- El paradigma conexionista tiene un fuerte carácter numérico. Las salidas numéricas de la RNA se interpretan en términos de las etiquetas asociadas a las *clases* de salida.

Esta aproximación del conexionismo nos habla de una red de estructura fija, que actúa como un clasificador. Interpreta el sistema como un mecanismo de adaptación de un agente a su medio, considerando así a la inteligencia como un medio superior de adaptación, construido sobre otros medios de adaptación más elementales. Se utiliza cuando no se sabe representar de forma explícita el razonamiento para la solución de un problema, por lo que se acude a modelos numéricos aproximativos cuyos valores se ajustan a base de la experimentación y el aprendizaje.

2.4. El paradigma híbrido

La mayoría de problemas en IA suelen ser de naturaleza híbrida. Por tanto, es lógico pensar en soluciones también híbridas, combinando los datos y el conocimiento disponible con elementos o técnicas de varios paradigmas distintos.

Por ejemplo, para el control de un robot, podemos necesitar aproximaciones reactivas y declarativas, combinando el paradigma situado, en el que el robot obtiene parte de la información mediante sus perceptores y actúa mediante sus efectores, con otros paradigmas, como el representacional, utilizando técnicas simbólicas y deliberativas, o el conexionista, empleando técnicas neuronales para clasificar datos.

Algunos criterios a la hora de combinar técnicas o métodos de diferentes paradigmas son:

- Analizar las exigencias computacionales del problema y el conjunto de recursos disponibles. Eso ya nos puede dar una idea de qué métodos o técnicas pueden ser los más adecuados para resolver la tarea.

- Si no es suficiente, debemos seguir descomponiendo la tarea en otras más simples, hasta que lleguemos a decidir de qué tarea disponemos del conocimiento suficiente para usar reglas simbólicas, y de cuál no tenemos suficiente conocimiento, por lo que deberemos usar redes neuronales, probabilísticas (bayesianas) o conjuntos borrosos.

- El siguiente paso es la operacionalización efectiva del esquema, usando módulos simbólicos y neuronales, resultado de las decisiones de la fase anterior.

T3 – Introducción a las técnicas de búsqueda.

En IA, la resolución de problemas y búsqueda se refiere a un conjunto de técnicas y métodos que se utilizan en diferentes dominios de aplicación, como la deducción, la elaboración de planes de actuación, razonamientos, etc.

3.1. Conceptos básicos

Un **sistema de búsqueda** tiene 3 componentes principales:

- El conjunto de estados, que representa todas las situaciones por las que el agente puede pasar durante la búsqueda de la solución del problema.

- Los operadores, que modelan las acciones elementales que es capaz de realizar el agente sobre el medio, y, además, cada operador debe tener un coste asociado, que puede ser arbitrario o atribuido por la propia naturaleza del problema.

- La estrategia de control, que es la responsable de decidir el orden en que se van explorando los estado. Una estrategia de control inteligente (*heurística o informada*) debería llevar a explorar primero aquellos nodos que están en el mejor camino hacia una solución óptima. Sin embargo, una búsqueda a ciegas (*no informada*) generalmente considerará todos los nodos como igualmente prometedores. Por tanto, con una búsqueda informada lo que se pretende es llegar a una buena solución (generalmente óptima) del problema, visitando el menor número de nodos posibles.

Los dos primeros (estados y operadores), conforman lo que se llama el *espacio de búsqueda*, que generalmente se representa mediante un grafo dirigido simple, en donde los nodos son los distintos estados por los que puede pasar el sistema, y los arcos son las reglas que provocan la transición. Generalmente, el grafo que representa el espacio de estados del problema tiene un tamaño tan grande, que no es posible representarlo de forma explícita, por lo que queda definido de forma implícita por un estado inicial y un conjunto de operadores. En este grafo, hay uno o varios nodos que representan soluciones del problema: son los llamados objetivos.

Un ejemplo de un sistema de búsqueda podría ser el caso de un robot que tiene un espacio de bloques, y, dada una situación inicial, tiene que conseguir colocar los bloques en una determinada posición. El conjunto de estados serían las distintas configuraciones de posiciones de los bloques, los operadores serían los movimientos que podría realizar el robot para trasladar los bloques, y la estrategia de control sería la definida por el diseñador, que podría ser no informada o informada.

Un algoritmo de búsqueda es **completo** si siempre encuentra una solución al problema de búsqueda, en el caso de que ésta exista. Un algoritmo de búsqueda es **admisible** (o exacto), si siempre encuentra una solución óptima.

3.2. Métodos de búsqueda sin información en árboles

El algoritmo general de búsqueda en árboles es como sigue: en primer lugar, se parte del nodo inicial (raíz del árbol), y a partir de ahí, el algoritmo va expandiendo nodos. En cada iteración, el nodo que se expande será el primero de una lista de posibles nodos candidatos a ser expandidos (*ABIERTA*), ordenada según el criterio que definan los distintos tipos de búsqueda en árboles: anchura, profundidad y coste uniforme. Este criterio de ordenación será el conocido como estrategia de control.

Búsqueda primero en anchura

En esta búsqueda, el criterio que define la ordenación de *ABIERTA* es insertar los nuevos nodos que se van generando simplemente al final de la lista, tratada como una cola FIFO.

Este algoritmo siempre es completo (para un grafo localmente finito, número de hijos por

nodo limitado). Además, si el coste de todas las reglas es 1, también es admisible. Sin embargo, tanto el tiempo de ejecución como el espacio de memoria necesario crecen de forma exponencial con el tamaño del problema. Este crecimiento está relacionado con la profundidad del árbol que representa el espacio de búsqueda.

Búsqueda primero en profundidad

Para esta búsqueda, el criterio de ordenación de la lista *ABIERTA* es el de insertar los nuevos nodos generados al principio de la lista, tratándola así como una pila LIFO.

El algoritmo de búsqueda en profundidad no es admisible, y ni siquiera es completo. Esto es debido a que la búsqueda puede derivar hacia una rama infinita, y el algoritmo no terminaría nunca. Por ello, se suele establecer una profundidad límite a partir de la cuál se detiene la búsqueda, se haya o no encontrado la solución.

En cuanto al espacio en memoria, este algoritmo tiene un coste lineal en relación a la profundidad, puesto que en cada hilo de ejecución solo almacena los nodos de una rama concreta, no los de todo el árbol. En relación al tiempo de ejecución, en el peor de los casos, este algoritmo tiene, como en la búsqueda en anchura, una complejidad exponencial a la profundidad del árbol. Además, la búsqueda en anchura siempre proporciona la solución más cercana al nodo inicial (la de menor altura en el árbol), lo que no siempre ocurre con la búsqueda en profundidad.

Búsqueda de coste uniforme

Este algoritmo no trata la lista *ABIERTA* como una cola ni una pila, sino que inserta los nodos en la lista, ordenados directamente por el coste desde el nodo inicial a cada uno de los nodos, de menor a mayor. Si el coste de todas las reglas es el mismo (por ejemplo, coste unitario – 1), la estrategia de coste uniforme y anchura son idénticas.

El coste computacional en espacio y memoria es similar al de la búsqueda en anchura (exponencial), y también es un algoritmo completo. Sin embargo, este algoritmo siempre obtiene la solución de menor coste al nodo inicial por lo que además es admisible.

Búsquedas en profundidad y anchura iterativas

Estas búsquedas se basan en limitar la profundidad o anchura límite (respectivamente), y realizar varias iteraciones del mismo algoritmo, incrementando dichos límites en cada iteración. Ambas versiones iterativas de dichos algoritmos tienen los mismos costes computacionales que sus respectivas no iterativas.

Sin embargo, en el caso de la búsqueda en profundidad se resuelve el problema de su profundidad límite (ramas infinitas) y en el caso de la búsqueda en anchura, ahora es posible encontrar una solución que no sea la más próxima al nodo inicial.

3.3. Métodos de búsqueda sin información en grafos

En los grafos, al contrario que en árboles, es posible que durante la búsqueda se pueda encontrar varias veces el mismo nodo *n* como sucesor de dos o más nodos diferentes. En este caso, se deben considerar los distintos caminos hasta dicho nodo *n*, y registrar solo el mejor obtenido hasta el momento. Por ello, puede ser necesario rectificar en la *TABLA_A* el coste desde el estado inicial hasta ese nodo *n*, así como su antecesor.

Algoritmo general de búsqueda en grafos (AGBG)

Este algoritmo parte de un grafo dirigido simple definido implícitamente a partir de un nodo inicial y una serie de operadores o reglas de producción. Cada regla tiene un coste no negativo. El grafo debe ser localmente finito (número de sucesores por nodo acotados), pero no necesariamente finito (se puede extender infinitamente en profundidad). En el grafo hay uno o varios estados solución y el objetivo de la búsqueda es encontrar el

camino de coste mínimo desde el nodo inicial hasta los estados objetivo.

Al expandir un nodo *n* determinado, hay que comprobar si alguno de sus sucesores, por ejemplo *s* ya fue expandido con anterioridad, para comprobar si dicho nodo *s* tiene un camino mejor mediante el nodo *n* que se acaba de expandir que mediante su camino anterior. Esto se hace con una función *Rectificar*, que se aplica a dicho nodo *s* en el caso de ya haber sido expandido con anterioridad, y también se aplica recursivamente a todos los sucesores de *s*, como por ejemplo un nodo *q*, porque puede que también mejore su camino de coste mínimo al nodo inicial pasando por *n* y por *s*. De esta forma, en la *TABLA_A* se registrará el mejor camino obtenido hasta el momento desde el nodo inicial a cada uno de los nodos encontrados durante la búsqueda en el punto actual.

La estrategia de control del AGBG queda definida por la forma de ordenar la lista *ABIERTA* de nodos candidatos a ser expandidos. Así, si ordenamos la lista insertando los nuevos nodos al principio de abierta (pila) tendremos una búsqueda en profundidad. Si ordenamos la lista insertando los nuevos nodos al final (cola), tendremos una búsqueda en anchura. Y por último, si ordenamos la lista *ABIERTA* mediante el coste de cada nodo al inicial, de menor a mayor, se obtiene una búsqueda de coste uniforme.

Búsqueda bidireccional

La idea de este algoritmo consiste en buscar simultáneamente en dos direcciones: por un lado, hacia delante desde el nodo inicial a los nodos objetivo, y por el otro lado, hacia atrás, desde los nodos objetivo hasta el nodo inicial. La búsqueda se detendrá cuando las listas *ABIERTA* de ambos recorridos tengan algún nodo en común. Esto consigue que el tiempo de ejecución, en lugar de ser exponencial en relación a una profundidad *d*, lo sea en relación a una profundidad *d/2*, lo que supone una mejora.

Sin embargo, hay varios factores que condicionan esta mejora. Por un lado, hay que tener claro cuántos y cómo son los estados objetivo, puesto que si son muchos, la mejora de eficiencia deja de ser cierta. Por otro lado, hay que considerar si los operadores son bidireccionales, puesto que si no lo son, a veces resulta difícil decidir cuales son los sucesores de un estado en el recorrido hacia atrás. Y por último, hay que decidir que algoritmo de búsqueda emplear en cada dirección, puesto que para asegurar que ambos recorridos se encuentren en algún momento, al menos uno de ellos debe almacenar en memoria todos los estados visitados, es decir, al menos uno debe ser en anchura.

T4 – Técnicas basadas en búsquedas heurísticas.

Los llamados *heurísticos* son aquellos mecanismos que permiten, en un espacio de estados, dirigir en cierta manera la búsqueda hacia las zonas más prometedoras, de modo que se pueda llegar a la solución sin necesidad de visitar tantos nodos como en general requeriría una búsqueda a ciegas o no informada.

Sin embargo, estos *heurísticos* también introducen un cierto coste de control, por lo que se debe alcanzar un punto de compromiso entre el coste de control y el coste de aplicación de las reglas, el cual es a veces difícil de alcanzar.

4.1. Búsqueda primero el mejor (BF)

En este algoritmo se emplea una función de evaluación de los nodos *f*, tal que para cada nodo *n*, la función *f(n)* da un valor numérico que indica lo prometedor que es ese nodo para ser expandido. Así, la lista *ABIERTA* se ordena en base a *f*, estando los nodos candidatos más prometedores al principio. Esta medida *f* de lo prometedor que es un nodo se denomina *función heurística de evaluación*, y se puede estimar de varias formas.

Este modo de expansión de nodos candidatos no siempre llevará de forma directa a la mejor solución, pero si permiten generalmente llegar a buena soluciones expandiendo un número de estados mucho menor que con una elección puramente aleatoria.

4.2. El algoritmo A*

A continuación se introducen algunos conceptos previos (consideremos un nodo *n*):

- ***g*(n)*** es el coste del camino más corto desde el nodo inicial a *n*.

- ***h*(n)*** es el coste del camino más corto desde *n* al nodo objetivo más cercano a *n*.

- **f*(n) = g*(n) + h*(n)**. Es decir, *f*(n)* es el coste del camino más corto desde el nodo inicial a los nodos objetivos condicionado a pasar por *n*.

- **C*** = f**(inicial)* = h**(inicial)*. Es decir, *C** es el coste de la solución óptima.

Todos los nodos que forman parte de la solución óptima cumplen que *f*(n) = C**, mientras que aquellos que no están en el camino óptimo cumplen que *f*(n) > C**. Para problemas complejos, los valores de estas funciones no se pueden conocer, por lo que el algoritmo A* lo que hace es trabajar con aproximaciones a estos valores.

Así, *g(n)* es el coste del mejor camino desde el inicial al nodo *n* obtenido hasta el momento, *h(n)* es una estimación positiva del valor de *h*(n)* tal que *h(n) = 0* si *n* es un nodo objetivo. Por último, *f(n) = g(n) + h(n)*, siendo *f* una estimación de *f**, y siendo el criterio que se utiliza para ordenar la lista *ABIERTA*.

Propiedades formales

- A* es completo en grafos localmente finitos (nº sucesores acotado).

- A* es admisible, siempre que utilice una función heurística *h* admisible -> *h(n) < h*(n)*.

- Un heurístico es monótono si para todo par de nodos *n* y *n'* se cumple que:

· *h(n) ≤ k(n, n') + h(n')*. Siendo *k(n, n')* el coste mínimo de *n* a *n'*. Infinito si no hay ruta.

- Todo heurístico que sea monótono, también es admisible.

- Si *h* es monótono y A* elige por ejemplo un nodo *n* para su expansión, se cumple que *g(n) = g*(n)*. Es decir, el camino del inicial a *n* encontrado hasta el momento es óptimo. Debido a ésto, si *h* es monótono, no es necesario rectificar nodos ya expandidos, por lo que el algoritmo A* es más eficiente utilizando un heurístico monótono.

Relajación de las condiciones de optimalidad

Se puede plantear conseguir más eficiencia con el algoritmo A*, a costa de perder la admisibilidad que proporciona. Existen dos métodos principales:

- Ajuste de los pesos de *g* y *h*. La búsqueda de A* está basada en la fórmula $f(n) = g(n) + h(n)$, por tanto, se pueden ponderar ambos sumandos, de tal forma que se obtenga una mayor eficiencia de la siguiente forma: $f_w(n) = (1-w)\cdot g(n) + w\cdot h(n)$, con $w \in [0,1]$. El mejor valor de *w* se puede obtener de forma experimental.

- Algoritmos ε-admisibles. Se sacrifica la obtención de una solución óptima en favor de alguna mejora en el rendimiento del algoritmo, controlando el deterioro de la solución obtenida a través de un factor ε que representa la distancia al coste óptimo.

4.3. Búsqueda con memoria limitada

El principal problema del algoritmo A* es el requerimiento de memoria, que crece de forma exponencial con la profundidad, aunque dispongamos de buenos heurísticos.

Algoritmo IDA (Iterative Deepening A*)*

Este algoritmo extiende al de búsqueda en profundidad iterativo, y se basa en realizar iteraciones de búsqueda primero en profundidad desde el nodo raíz, aumentando en cada iteración la profundidad límite de dichas búsquedas. En la primera iteración, se establece una longitud límite igual al valor de *f(inicial)*, descartando todos aquellos nodos cuya estimación *f(n)* supere dicha longitud límite. A continuación, si no se ha encontrado una solución, se realiza una nueva iteración en la que la búsqueda comienza otra vez desde el principio, pero ahora la longitud límite será el menor valor de *f* de entre todos los nodos descartados en la iteración anterior (aumentando la profundidad, por tanto).

Los hijos de cada nodo expandido se introducen ordenados en *ABIERTA* según su valor *f*, actuando *ABIERTA* como una pila. Así, se consideran antes los hijos más prometedores, y, en caso de encontrar una solución, se habrán expandido menos nodos.

Si *h* es admisible, el algoritmo *IDA** que lo utilice también lo será. Por otro lado, el consumo de memoria de este algoritmo es proporcional al producto de la profundidad de la solución y del factor de ramificación (nº de hijos por nodo), lo que supone un ahorro de memoria. Sin embargo, el tiempo de búsqueda es exponencial con la profundidad límite.

Algoritmo SMA (Simplified Memory-Bounded A*)*

En este caso, el algoritmo se basa en limitar el tamaño de la *TABLA_A*, es decir, el máximo nº de nodos que se pueden almacenar en ella. Si se necesita expandir un nodo, y no hay espacio suficiente en la *TABLA_A*, se elimina un nodo de *ABIERTA* y de la *TABLA_A*, aquel con mayor valor de *f* en *ABIERTA*, el cual se conoce como *nodo olvidado*. El algoritmo recuerda en cada nodo el mejor *f* de los hijos de ese nodo (los nodos olvidados). De este modo, solo reexplora un subárbol descartado si el resto de posibilidades tiene estimaciones que son aún peores.

Este algoritmo adaptativo es capaz de evolucionar según la memoria disponible. Será completo si la memoria disponible es suficiente para almacenar el camino a la solución menos profunda. Además, es admisible si tiene suficiente memoria para almacenar el camino hasta la solución óptima menos profunda; si no, devuelve la mejor solución encontrada con la memoria disponible.

4.4. Algoritmos voraces

Estos algoritmos se basan en la idea básica de tomar decisiones de forma irrevocable. Es decir, los nodos que han sido descartados no los vuelve a considerar. Por tanto, no son admisibles, y tampoco suelen ser completos, pero a cambio resultan muy eficientes, por lo

que se suelen usar en aplicaciones de tiempo real (planificación del procesador).

El diseño de este tipo de algoritmos es simple. Por ejemplo, si partimos del algoritmo A*, y en cada paso consideramos solo el nodo más prometedor en función de *h*, descartando el resto de sucesores, obtenemos un algoritmo voraz. Si, además, los empates que se puedan producir en cada paso se resuelven de forma aleatoria, tenemos un algoritmo no determinista, pues en cada ejecución del mismo sobre un problema, nos puede dar resultados diferentes, aunque con una eficiencia media que dependerá de *h*.

4.5. Algoritmos de ramificación y poda

Los algoritmos de ramificación y poda interpretan cada estado o nodo como un subconjunto de soluciones de todo el conjunto posible de soluciones del problema original planteado. De esta manera, el nodo raíz representa todas las soluciones posibles al problema original, mientras que un nodo hoja sería aquel que no puede ser expandido por contener una única solución. Así, el proceso de ramificación consiste en descomponer un determinado conjunto de soluciones (un nodo) en la unión disjunta de varios subconjuntos suyos (los nodos sucesores), con lo que el espacio de búsqueda adquiere forma de árbol.

Mientras que un nodo hoja tendrá asociado un coste concreto y conocido, cada nodo intermedio del árbol tiene asociado un valor heurístico que representa una cota inferior del coste de la mejor solución (la de menor coste) contenida en el nodo. Generalmente, cada vez que la cota inferior de un nodo rebasa el menor de los costes de los nodos hoja encontrados hasta el momento en el resto del árbol, dicho nodo es podado.

Para ahorrar espacio en memoria, generalmente la estrategia de control para la exploración del árbol suele ser en profundidad. En este caso, *ABIERTA* actúa como una pila y se introducen en ella los nodos generados en cada expansión ordenados según los valores de sus cotas inferiores (de menor a mayor). La eficiencia del algoritmo de ramificación y poda dependerá de lo ajustadas que sean las cotas inferiores obtenidas de forma heurística. Por otra parte, los algoritmos de ramificación y poda son admisibles si recorren todo el espacio de búsqueda, excepto las partes podadas, hasta que *ABIERTA* se agote; en caso contrario, si la condición de terminación es que el algoritmo pare después de expandir cierto número de nodos, se pierde la admisibilidad, ganando eficiencia, y únicamente se puede devolver la mejor solución encontrada hasta el momento.

4.6. Algoritmos de mejora iterativa o búsqueda local

Existen problemas donde lo que interesa no es el camino desde el nodo inicial hasta el nodo objetivo, sino que el nodo objetivo ya contiene de por si toda la información necesaria, y el camino es irrelevante. Es el caso de los algoritmos de búsqueda local.

El algoritmo parte de una solución inicial, y en cada iteración calcula un conjunto de *soluciones vecinas* mediante una regla de vecindad. Cada una de estas soluciones vecinas es evaluada, y *se selecciona una* de ellas con un criterio, que suele ser elegir la de menor coste. Si la solución escogida cumple el *criterio de aceptación* (normalmente ser mejor que la solución actual *S*), la solución elegida reemplaza a dicha solución *S*. Así, el proceso continúa hasta que se cumple el *criterio de finalización*, que generalmente es agotar un número *n* de iteraciones, o que no se produzcan mejoras en los últimos *n* intentos.

Algoritmo máximo gradiente

En este tipo de búsqueda, el criterio de aceptación es que la solución vecina S_1 encontrada sea mejor o igual que la solución *S* actual. Es una búsqueda simple, pero que tiene algunos problemas: óptimos locales (si se alcanza uno de ellos, todos los vecinos serán peores, y el algoritmo finaliza sin encontrar el óptimo global); regiones planas (todos los vecinos tendrán el mismo valor que la solución actual, y la búsqueda es aleatoria); y crestas (si la pendiente es suave, resulta difícil no desviarse hacia los lados al ascender).

Debido a esos problemas, la búsqueda puede quedarse atascada. Una solución puede ser

reiniciar la búsqueda a partir de otro punto de inicio, lo que se denomina *búsqueda multiarranque*. Con este método se alcanzaría probablemente otro óptimo local distinto, y así, después de varios arranques se alcanzaría una solución aceptable.

Temple simulado

Este algoritmo realiza una selección aleatoria entre los vecinos del estado actual. Si el nodo seleccionado mejora la solución actual, la búsqueda sigue como en el caso de máximo gradiente. Si no, existe una cierta probabilidad de que dicho nodo sea aceptado, aún cuando sea peor que la solución actual, consiguiendo eludir máximos locales, si los hubiera. Dicha probabilidad depende de: la temperatura T, y el incremento de energía ΔE, que es la diferencia entre el coste de la nueva solución, y el coste de la solución actual.

Al principio la temperatura tiene un valor alto, de modo que la probabilidad de aceptar un nodo cuya solución es peor es elevada, favoreciendo así la exploración del espacio de soluciones. A medida que avanza, la temperatura va decreciendo, por lo que al final del algoritmo, se da preferencia a la búsqueda de soluciones de calidad, convergiendo hacia soluciones que siempre mejoren a la actual. El principal problema de este algoritmo es la elección de un enfriamiento adecuado, el cual depende de la naturaleza del problema.

Búsqueda tabú

Lo que caracteriza a este algoritmo es que dispone de una memoria, denominada *lista tabú*. En base a dicha lista, se puede evitar la generación de vecinos que conduzcan a solución no óptimas o ya revisadas, ahorrando tiempo y mejorando la eficiencia.

En algunos casos, se pueden aplicar excepciones a dicha lista, utilizando lo que se conoce como *criterio de aspiración*, que consiste en admitir nodos tabú que puedan mejorar la solución actual, expandiendo dichos nodos como si no estuviesen en la lista.

El principal inconveniente de este algoritmo es el ajuste de los parámetros, como el tamaño de la lista tabú, el criterio para incluir un nodo en dicha lista, y la definición concreta del criterio de aspiración, para permitir excepciones. Dichos parámetros son fuertemente dependientes del problema, por lo que se deben particularizar a cada uno.

T7 – Redes semánticas.

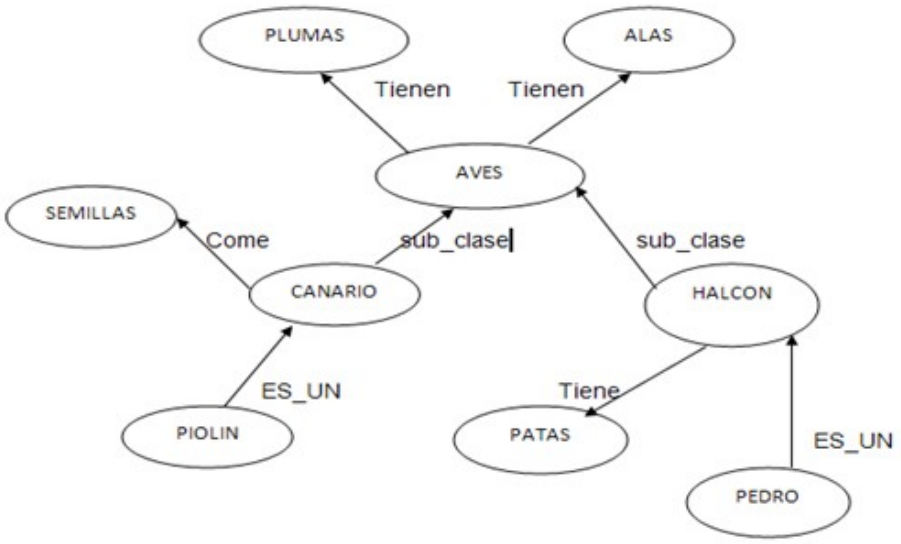
Formalizar consiste en representar simbólicamente los conocimientos de un dominio utilizando alguno de los formalismos de representación de conocimientos existentes (paso del *modelo conceptual* al *modelo formal*). Hay distintos tipos de formalismos: basados en conceptos (*marcos*), basados en relaciones (*redes semánticas*) y basado en acciones.

7.1. Representación del conocimiento

Las **redes semánticas** son un formalismo o paradigma de representación de conocimiento basado en relaciones entre los conceptos o entidades de un dominio.

Representación básica

En este formalismo, la información se representa en un grafo dirigido formado por un conjunto de *nodos* y *arcos unidireccionales*, ambos etiquetados. Los nodos representan conceptos e instancias de dichos conceptos, y los arcos conectan los nodos y representan relaciones binarias entre ellos (predicados de aridad dos). Ejemplo:



La base de la representación consiste en modelar conocimientos relativos a un objeto (concepto) mediante pares *atributo-valor*. Así, el nodo origen es el *concepto*, el arco que los une es el *atributo*, y el nodo destino es el *valor* para dicho atributo de ese objeto.

Los arcos en las redes conceptuales (semánticas) se agrupan en dos categorías:

- Arcos descriptivos. Describen entidades y conceptos. Por ejemplo, un arco descriptivo para una red que representase de alguna forma personas podría ser *Profesión*. En el ejemplo superior un arco descriptivo es *Come*, que indica que un *Canario Come Semillas*.

- Arcos estructurales. Enlazan las entidades o conceptos formando la arquitectura o estructura de la red. La semántica de estos arcos es independiente del dominio del problema. Se pueden definir tantas etiquetas estructurales como se quiera:

  · *Generalización*, ponen en relación una clase con otra más general. Las propiedades definidas en los nodos generales son heredadas por deducción por los nodos específicos, mediante arcos *Subclase-de*.

  · *Arco instancia*, liga un objeto con su tipo genérico. Se llama arco *Instancia (ES_UN)*.

  · *Agregación*, liga un objeto con sus componentes. Se llama arco *Parte-de (inverso de Tiene/n)*.

Los conocimientos expresados en una red semántica también pueden expresarse utilizando lógica proposicional y cálculo de predicados de primer orden.

Representación de predicados no binarios

Generalmente, en las redes semánticas solo es posible representar predicados de aridad dos. Para representar predicados de aridad 3 o superior, es necesario crear un nuevo objeto que represente al predicado de aridad mayor que dos, y definir nuevos predicados binarios que describan las relaciones entre este nuevo objeto y sus argumentos.

Por ejemplo, si quisiéramos representar el predicado cuaternario *COMPRA-VENTA(Pepe, Luis, Reloj, 45)*, no podríamos hacerlo directamente. Para ello, crearíamos un nuevo objeto *COMPRA_VENTA_1*, instancia del objeto genérico *COMPRA_VENTA*, y crearíamos varios predicados binarios que unieran el nuevo objeto creado *COMPRA_VENTA_1* con sus atributos y valores, tal que así: arco *Comprador* con valor *Pepe*, arco *Vendedor* con valor *Luis*, arco *Objeto* con valor *Reloj* y arco *Precio* con valor *45*.

Representación de acciones

Se basa en la *gramática de casos*. En ella, toda proposición tiene una estructura formada por un verbo y una o varias frases nominales. Cada frase nominal se relaciona con el verbo mediante un conjunto de casos, que pueden ser: *agente* (persona que realiza la acción), *contra-agente* (resistencia contra la que se ejecuta la acción), *objeto* (entidad cuya posición o existencia se considera), *lugar* (en el que se desarrolla el evento), *tiempo* (fecha o momento concreto) y *sujeto* (entidad que sufre el efecto).

La modalidad por su parte hace referencia a características que presenta el verbo, como por ejemplo: *tiempo* (pasado, presente o futuro) y *voz* (activa o pasiva).

Se utiliza esta información proporcionada por la gramática de casos para representar afirmaciones que se refieren a acciones y eventos. Cada nodo *situación* tiene como atributos el conjunto de casos y de modalidades que describen el evento. Por ejemplo: acción *VER_1* (instancia de *VER*), con atributos de *voz, tiempo, lugar, objeto*, etc.

Representación de conocimiento disjunto

Para representar entidades del domino que son disjuntas entre sí (no tienen elementos comunes), se puede utilizar simplemente el arco *Disjunto* de una entidad a otra.

Sin embargo, existe otro formalismo que utiliza arcos especiales, como sigue: *arco S* (subconjunto), *arcoSD* (subconjunto disjunto), *arco E* (elemento) y *arco ED* (elemento disjunto). Por ejemplo, tendríamos una entidad *Ser-Vivo* que representa el conjunto de seres vivos, la cual tiene dos subconjuntos *Plantas* y *Animales* disjuntos entre sí. Por tanto, tendríamos un arco *SD* desde la entidad *Plantas* y otro arco *SD* desde la entidad *Animales* que irían ambos hacia la entidad *Ser-Vivo*.

7.2. Inferencia de conocimiento

Para resolver los casos que se planteen, en una red semántica se deben utilizar procedimientos que trabajen con la semántica de sus arcos. Así, tenemos dos técnicas diferentes según su forma de proceder:

Equiparación

Se dice que un *apunte* (fragmento de una red) se equipara con la red semántica si el apunte puede asociarse con un fragmento de la red semántica. Los pasos a seguir son:

- Paso 1. Se construye un apunte que responda a la pregunta que se quiere resolver, formado por un conjunto de nodos *constantes* (datos conocidos de la pregunta), nodos *variables* (valores que se requieren, son desconocidos) y *arcos etiquetados* (como arco *agente*, arco *lugar*, arco *objeto*, etc, que unen nodos constantes y variables).

- Paso 2. A continuación, se coteja el apunte con la red semántica.

- Paso 3. Los nodos variables del apunte se ligan a nodos constantes de la red semántica hasta encontrar una equiparación perfecta.

- Paso 4. La respuesta a la consulta es el fragmento de red semántica con los valores con los que se rellenan los nodos variables. Ejemplo: *Ver-? = Ver_1* y *Varón-? = Pepe*.

Herencia de propiedades

Permite que nodos específicos de una red accedan a las propiedades definidas en otros nodos utilizando los arcos *Instancia* y *Subclase-de*, evitando así la redundancia de propiedades en la base del conocimiento.

Para determinar la veracidad de una sentencia cualquiera sobre un atributo o propiedad de una entidad, se debe localizar el nodo entidad al que se hace referencia, y comprobar si desde dicho nodo sale un arco con la etiqueta del atributo evaluado. Si no existiese dicho arco, el sistema buscará arcos *Instancia* que partan desde dicho nodo hacia algún otro, y así repetidamente empleando arcos *instancia* o *Subclase-*de hacia las entidades superiores más generales. Si una vez se han explorado todas las alternativas, no se encuentra la solución, entonces se debe comunicar que, con la información almacenada en la red semántica, no se puede contestar sobre la verdad o falsedad de la sentencia inicial.

La distribución de propiedades en la red permite que se herede el valor de la propiedad del nodo más cercano al nodo que sirvió como punto de partida en la inferencia. Así, ante la posibilidad de heredar un valor de dos nodos distintos, se hará del más cercano.

Los principales errores que se suelen cometer utilizando herencia de propiedades son:

- No distinguir bien los nodos que son instancias de aquellos que son conceptos.

- Que el nombre etiquetado tenga una semántica diferente al conocimiento representado.

- Establecer arcos en sentido contrario al natural o adecuado.

- No representar situaciones empleando nodos situación o evento.

T8 – Marcos.

Son la técnica de representación del conocimiento más utilizada cuando éste *se basa en conceptos*. El conocimiento que expresan es declarativo. Los marcos organizan los conocimientos del dominio en árboles (jerarquía) o en grafos, ambos construidos por especialización de conceptos generales en otros más específicos.

8.1. Representación de conocimiento

Los conceptos básicos al formalizar la base de conocimientos son: ***marcos***, para representar conceptos o elementos, ***relaciones***, para expresar dependencias entre conceptos, ***propiedades***, para describir cada concepto, y ***facetas***, para expresar de múltiples formas los valores con los que se puede rellenar cada propiedad.

Representación de conceptos e instancias

De manera general, existen dos tipos de marcos:

- Los marcos clase. Se utilizan para representar conceptos, clases o situaciones genéricas descritas por un conjunto de propiedades, unas con valores y otras sin valores asignados, que son comunes al concepto que el marco representa. Ejemplo: marco *Persona*.

- Los marcos instancia. Pueden considerarse como la representación en el dominio real de una clase determinada. Deben estar relacionados, como mínimo, con un marco clase. Además, suelen rellenar la mayoría de sus propiedades con valores específicos de la instancia que representan. El resto de propiedades las hereda de los marcos clase de los cuales son instancias. Ejemplo: marco *Juan* (instancia del marco clase *Persona*).

Representación de relaciones entre conceptos

El formalismo de marcos representa las relaciones del dominio mediante relaciones entre marcos clase, marcos instancia, y marcos clase y marcos instancia, formando así un sistema basado en marcos (SBM). Existen diferentes tipos:

- **Relaciones estándar**. Son independientes del dominio. Hay varios subtipos:

  · Relación *subclase-de*. Se define entre marcos clase. Permite construir un SBM mediante la especialización de conceptos generales en conceptos más específicos. Su inversa es la relación *superclase-de*. A un marco clase pueden llegar y/o partir de él un número indefinido de relaciones de este tipo (herencia múltiple).

  · Relación *instancia*. Se define entre un marco instancia y un marco clase. Representa que el marco instancia es un elemento del conjunto o clase representado por el marco clase. Un elemento puede pertenecer a varios conjuntos simultáneamente; por tanto, de un marco instancia pueden partir tantas relaciones instancia como conceptos describan el marco consistentemente. Su inversa es la relación *representa*, y va del marco clase al marco instancia.

- **Relaciones no estándar**.

  · Relación *fraternal*. Se define para dos marcos clase que tienen el mismo marco clase padre. Representa que dos conceptos son hermanos.

  · Relación *disjunto*. Se define para dos marcos clase. Representa que las clases son disjuntas, es decir, que los conjuntos que representan ambos marcos no tienen elementos en común. Su inversa es la relación *no-disjunto*, que representa marcos clase conectados los cuales si pueden tener elementos comunes.

  · Relaciones *ad-hoc*. Sirven para representar relaciones 'a medida' entre conceptos de un dominio. Se debe comprobar previamente que: *A)* la relación *ad-hoc* haya sido definida previamente entre dos marcos clase, *B)* que los marcos instancia

que se quiere conectar sean instancias de dichos marcos clase. Cada relación *ad-hoc* entre dos instancias es una instancia de la definida a nivel de marco clase.

Para conectar marcos clase de diferentes jerarquías, se debe comprobar además que se cumplen las restricciones propias de la relación entre jerarquías (ej: jugadores de fútbol y equipos). Algunas herramientas no implementan estas relaciones no estándar (se debe usar algún *truco* para representarlas - otro marco).

Representación de las propiedades de los conceptos

Existen dos tipos de propiedades a formalizar:

- Las *propiedades de **clase***. Representan atributos o características genéricas de un concepto o clase. Estas propiedades se rellenan en el propio marco clase, y toman siempre el mismo valor en todos los elementos o instancias de la clase.

- Las *propiedades de **instancia***. Aunque se definen en el marco clase y son comunes a todas las instancias de dicho marco clase, se rellenan en cada marco instancia con valores concretos que dependen del elemento. Van precedidas del *símbolo* '*' para distinguirlas.

En los marcos de clase, las propiedades de clase se rellenan con el valor que toma la propiedad, y las propiedades de instancia con el tipo de valor con el que éstas se pueden rellenar en las instancias (con un tipo de datos: entero, carácter, etc, o bien con un puntero a otro marco clase). Como con los tipos primitivos y punteros a objetos en Java.

Representación de facetas de propiedades

Las facetas permiten modelar características de las propiedades y relaciones en los marcos clase. El motor de inferencia usa las facetas para mantener la integridad semántica de los datos, es decir, para comprobar que los valores introducidos en las propiedades realmente pertenecen al tipo especificado. Hay 3 categorías:

- ***Facetas que definen propiedades de clase, de instancia y relaciones***.

  - Tipo ranura. Establece el tipo de datos con el que se rellenará la propiedad o relación. Existen tres casos diferentes: *propiedades de clase o instancia que se rellenan con valores* (aquí se especifica el tipo correspondiente), *propiedades de clase o instancia definidas como marcos* (aquí se especifica que se trata de un marco), y *relaciones* (se definirán en el marco clase origen de la relación, tendrán como nombre la relación, y se especificará que se trata de un marco).

  - Cardinalidad mínima. Establece el número mínimo de valores con los que se rellena la ranura, siempre que ésta se rellene.

  - Cardinalidad máxima. Establece el número máximo de valores con los que se puede rellenar esta ranura.

  - Multivaluada. Establece si la propiedad puede tener más de un valor o no.

- ***Facetas que definen propiedades de clase y relaciones***.

  - Propiedad general. Almacena los valores que toma una propiedad de clase o una relación. Las propiedades de clase definidas como marcos y las relaciones rellenan esta faceta con un puntero a un marco clase. Las propiedades de instancia nunca rellenan esta faceta, y suelen utilizar el símbolo "--" para indicarlo.

- ***Facetas que definen propiedades de instancia***.

  - Valores Permitidos. Especifica el conjunto de valores válidos que puede tomar una propiedad de instancia, el cual debe ser consistente con el contenido de la faceta "tipo ranura". Puede almacenar un tipo de datos, un rango de valores o un puntero.

- Valores por Omisión. Fija el valor que toma la propiedad de instancia en un marco instancia si no se especifica otro. Puede ser anulado al asignar un nuevo valor.
- Si Necesito. Almacena un procedimiento que se ejecuta al solicitar el valor de una propiedad de instancia y ser desconocido dicho valor. La ejecución de este procedimiento puede tomar datos de otras ranuras o del usuario del sistema.
- Si Modifico. Almacena un procedimiento que se ejecuta al modificar el valor de una propiedad de instancia. Su ejecución puede afectar a otras ranuras.
- Si Añado. Almacena un procedimiento que se ejecuta al introducir un valor en una propiedad de instancia que estaba vacía. Puede afectar a otras ranuras.
- Si Borro. Almacena un procedimiento que se ejecuta al borrar el valor de una propiedad de instancia. Puede afectar a otras ranuras.

8.2. Criterios de diseño

- Se debe favorecer la compartición de propiedades de clase y de instancia entre marcos.

- Se debe evitar representar conocimientos redundantes.

- Debido al carácter local de las propiedades, se pueden tener propiedades repetidas con el mismo nombre en diferentes marcos de clase.

- Se pueden redefinir las propiedades de clase/instancia en marcos clase más específicos.

- En un marco instancia se pueden rellenar, o no, todas las propiedades de instancia definidas en los marcos clase con los que está conectado.

- En las instancias no se pueden utilizar propiedades no definidas en los marcos clase.

8.3. Inferencia de conocimiento

El formalismo de marcos permite realizar inferencias utilizando 3 técnicas distintas:

Equiparación

Equiparar significa clasificar. Conocidos los valores de un conjunto de propiedades que describen parcialmente una nueva entidad o marco pregunta, esta técnica clasifica el marco pregunta en el grafo que representa el dominio. Se basa en encontrar los marcos clase de la BC que describen más consistentemente el marco pregunta, y éste último se convierte en una instancia de dichos marcos clase.

Es una técnica útil en SBM que clasifican o en sistemas que se enfrentan a situaciones parecidas a otras que ocurrieron anteriormente. Se descompone en tres etapas:

- *Selección de los marcos candidatos*. Si el tipo de la nueva entidad es conocido, se puede seleccionar el marco clase en el que se ha definido el tipo, y todos los marcos clase en los que éste se ha especializado. Si el tipo de la nueva entidad es desconocido, la selección de marcos clase se realiza arbitrariamente, o se eligen aquellos marcos clase en los que, como mínimo, se encuentre definida una propiedad conocida en el marco pregunta.

- *Cálculo del valor de equiparación (VE)*. Se calcula el VE del marco pregunta en cada uno de los marcos candidatos. El VE es una medida que informa del grado de idoneidad de la equiparación que se va a realizar. El cálculo de este VE varía de unas aplicaciones a otras.

- *Elección de los marcos clase con los que se equiparará la nueva entidad*. Para un marco clase determinado, si el valor VE es suficientemente alto, el sistema no buscará otros marcos e instanciará la nueva entidad convirtiéndola en un marco instancia de dicho marco clase. Si el valor VE no es lo suficientemente alto, se tendrán que identificar otros marcos relevantes buscando en el resto de la jerarquía (vertical u horizontalmente).

Herencia de propiedades

Permite compartir valores y definiciones de propiedades entre marcos de una BC usando para ello las relaciones *instancia* y *subclase-de*. Se puede distinguir entre:

- Herencia simple.

Se aplica cuando solo existe un único camino que une el marco instancia con el nodo raíz de la jerarquía (forma de árbol). El algoritmo para encontrar los valores de una cierta propiedad en un marco instancia es:

1) Se busca la propiedad en el marco instancia. Si se encuentra, se devuelven sus valores y fin, si no, se accede al marco clase padre utilizando la relación *instancia*.
2) Se busca la propiedad en el marco clase. Si se encuentra, se devuelven sus valores y finaliza, si no, se utiliza la relación *subclase-de* para acceder al marco clase padre, mientras éste no sea el marco raíz del árbol.
3) Se busca la propiedad en el marco raíz. Si se encuentra, se devuelven sus valores y finaliza, si no, debe responder que con la BC actual es imposible responder.

- Herencia múltiple.

Se aplica cuando existen varios caminos que unen los marcos instancia con el nodo raíz de la jerarquía (forma de grafo). Existen diferentes algoritmos que recorren el grafo:

- *Búsqueda en profundidad*. Explora en profundidad todos los posibles caminos que van desde el marco instancia al marco raíz del SBM. Algunos criterios para realizar el recorrido son: recorrer el grafo *de izquierda a derecha*, usar el criterio de *exhaustividad* (solamente se buscará la propiedad en cada marco una vez), y usar el criterio de *especifidad* (solo se puede buscar la propiedad en una clase si previamente se ha buscado en todas sus subclases).
- *Búsqueda en amplitud*. Recorre el grafo por niveles que están a igual distancia del marco instancia. Primero se busca la propiedad en los padres del marco instancia, si no la encuentra, se busca en los abuelos, y así sucesivamente. El proceso termina al encontrar la propiedad o alcanzar el nodo raíz sin encontrarla.
- *La distancia 'inferencial'*. Se puede definir como: *la condición necesaria y suficiente para que la clase$_1$ esté más cercana a la clase$_2$ que a la clase$_3$, es que la clase$_1$ tenga un camino de inferencia a través de la clase$_2$ hacia la clase$_3$*. Es decir, que la clase$_2$ esté en medio de la clase$_1$ y la clase$_3$.

Valores activos

Llamados *demonios* o *disparadores*, son procedimientos que recuperan, almacenan y borran información en los SBM. Se definen en las facetas *Si Necesito, Si Añado, Si Modifico* y *Si Borro* de las propiedades de instancia de los marcos clase. Características:

- Se definen en el marco clase y permanecen latentes hasta que se solicite su ejecución desde un marco instancia, momento en que el procedimiento asociado se ejecuta con los valores almacenados en las propiedades del marco instancia.
- Estos procedimientos pueden ser: demonios dirigidos por eventos (ejecutan procedimientos antes de almacenar o borrar valores en las propiedades de un marco instancia – asociados con las facetas *Si Añado, Si Modifico* y *Si Borro*), y demonios dirigidos por metas (deducen valores de propiedades a partir de valores almacenados en otras propiedades – asociados con la faceta *Si Necesito*).
- El control de ejecución va pasando de unas propiedades a otras a medida que se van ejecutando los procedimientos y produciéndose llamadas entre los mismos.