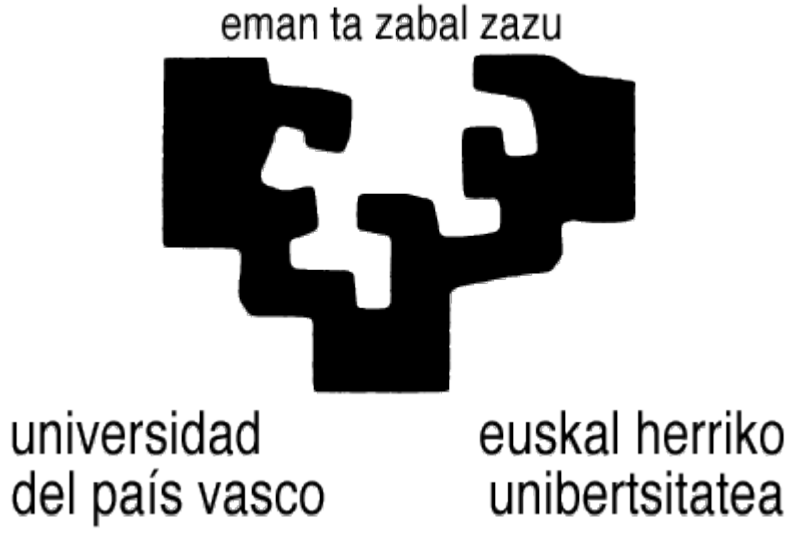
Facultad de Informática Informatika Fakultatea



TITULACIÓN: Ingeniería Informática

SSI-Dijkstra-Fast: Desarrollo de un sistema de resolución de la ambigüedad semántica basada en grafos.

Alumno/a: D./Dña. Gorka Blanco Gutierrez

Director/a: D./Dña. German Rigau Claramunt

Proyecto Fin de Carrera, julio de 2013

Índice de contenido

1.- Introducción

1.1.- Presentación

Este documento es la memoria del Proyecto de Final de Carrera realizado por Gorka Blanco Gutierrez (estudiante de Ingeniería Informática en la Facultad de Informática de San Sebastián, Universidad Pública Vasca (UPV/EHU)) y dirigido por Germán Rigau Claramunt (profesor de la Facultad de Informática de San Sebastián, Universidad Pública Vasca (UPV/EHU)).

El proyecto se encuadra en el campo de la Inteligencia Artificial. Más concretamente en el campo del procesamiento del lenguaje natural, de la semántica y de la desambiguación de los sentidos de las palabras (*Word Sense Disambiguation* en inglés).

1.2.- Motivación

1.2.1.- El problema de la desambiguación de palabras

La desambiguación de palabras (en inglés, WSD de *Word Sense Disambiguation*) se ha considerado desde siempre uno de los problemas más difíciles del Procesamiento del Lenguaje Natural. Para tratar de dar solución a este problema se han utilizado técnicas muy diferentes. En nuestro caso, vamos a centrarnos en el estudio de técnicas que usan grandes bases de conocimiento léxico. De forma aún más concreta, nos basaremos en un método de desambiguación basado en grafos llamado SSI-Dijkstra [Cuadros & Rigau, 2007], y en una mejora del mismo llamado SSI-Dijkstra-Fast.

1.2.2.- Un ejemplo

Imaginemos que disponemos de una lista de palabras en inglés, (por ejemplo: *fly*, con 20 significados posibles; *airport,* con 1 sólo significado posible; *plane,* con 9 significados posibles...) cuyo significado correcto desconoce el ordenador a priori. Como sabemos, las palabras pueden ser monosémicas (tienen un único significado, por ejemplo: *airport*) o pueden ser polisémicas (tienen varios significados, por ejemplo: *fly,* pudiendo referirnos a una mosca o al propio verbo volar). La única información adicional que tenemos de estas palabras es si son nombres, verbos, adjetivos o adverbios. Por ejemplo, *pilot* puede actuar al mismo tiempo como nombre (*piloto*) o como verbo

(*pilotar*). El problema es hallar el significado más adecuado para cada una de estas palabras cuando estas palabras coocurren en un mismo contexto. Al proceso que seguimos se le llama desambiguación.

Con el ejemplo, anterior, una posible solución coherente al problema podría ser la siguiente:

(*fly* – verbo → *travel through the air; be airborne*)
(*airport* – nombre → *an airfield equipped with control tower and hangars as well as accommodations for passengers and cargo*)
(*plane* – nombre → *an aircraft that has a fixed wing and is powered by propellers or jets*)
(*plilot* – verbo → *operate an airplane*)

1.2.3.- Objetivos del proyecto

Aunque será necesario profundizar en ello más adelante, el proyecto desarrollará el algoritmo SSI-Dijkstra-Fast (una versión del SSI-Dijkstra) basándose en implementaciones del algoritmo existentes para versiones anteriores de UKB. UKB es una herramienta de desambiguación semántica basada en grafos.

2.- Antecedentes

Los siguientes apartados explican brevemente el marco de trabajo del proyecto de final de carrera.

En primer lugar, presentaremos las bases de conocimiento léxico que usa el algoritmo de desambiguación que desarrollaremos en el proyecto. Nos centraremos en WordNet [Miller et al., 1990], la base de conocimiento léxico más usada en los algoritmos de desambiguación basados en el conocimiento. También presentaremos el proyecto europeo EWN (EuroWordNet)[Vossen, 1998]. y el MCR (Multilingual Central Repository) y los algoritmos SSI-Dijkstra, SSI-Dijkstra+ y SSI-Dijkstra-Fast [Cuadros & Rigau, 2007]; así como un software llamado UKB [Agirre & Soroa, 2009] que nos facilitará el desarrollo. Al final de todo, haremos un pequeño resumen de las diferentes etapas por las que ha pasado el algoritmo SSI-Dijkstra y sus diferentes aproximaciones.

2.1.- Redes semánticas

Una red semántica léxica es una forma de representación de conocimiento en la que los conceptos y sus interrelaciones se representan mediante un grafo. Los conceptos o unidades léxicas se representan mediante los nodos del grafo, y las interrelaciones que existen entre conceptos serán representadas por las aristas dirigidas del grafo.

En las redes semánticas, un concepto importante es la distancia semántica entre nodos del grafo que se expresa en relación al número y tipo de enlaces que separan un nodo de otro.

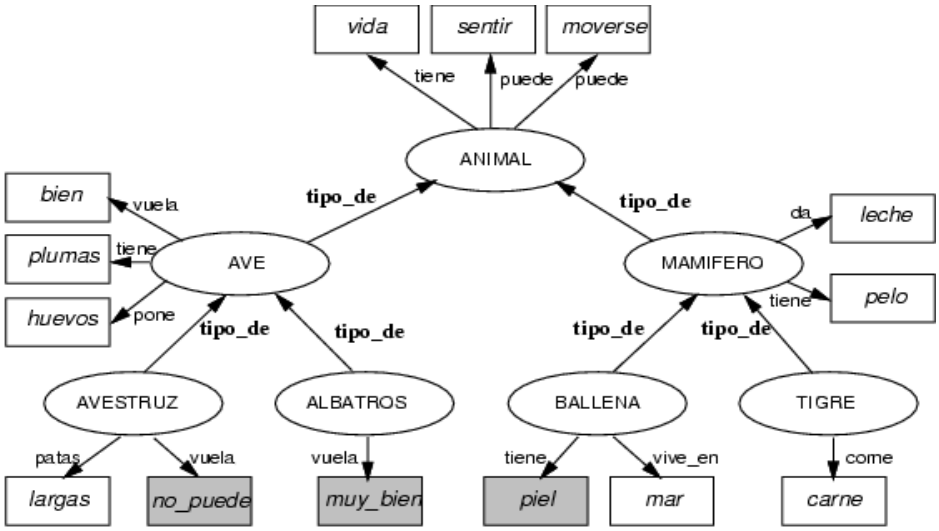


Imagen 1: Ejemplo de red semántica

En la Imagen 1 podemos ver representado cierto conocimiento acerca de los animales. En ella identificamos los elementos característicos de las redes semánticas como los nodos (mamífero, animal, ave, albatros, largas…) y las relaciones (tipo_de, vuela, tiene...).

Existen diversos tipos de relaciones semánticas como la sinonimia/antonimia (relación entre sinónimos y antónimos), hiponimia/hiperonimia (relación entre subordinados y superordinados), la meronimia/holonimia (relación entre subconjuntos y conjuntos globales), entre otras.

Una de las redes semánticas léxicas más conocidas es WordNet.

2.2.- Wordnet

Desarrollada en la universidad de Princeton, WordNet[1] (WN) [Miller et al., 1990] [Fellbaum, 1998] es una base de datos léxica de dominio general para el inglés que actualmente constituye uno de los recursos léxicos mas utilizados en el área de Procesamiento del Lenguaje Natural. WordNet fue creada para intentar organizar la información léxica por significados. A diferencia de los diccionarios convencionales, esta información está organizada conceptualmente. Actualmente, WordNet cuenta con 155.287 palabras, organizadas en 117.659 nodos para un total de 206.941 emparejamientos palabra-nodo.

WordNet está estructurada como una red semántica cuyos nodos, denominados synsets

1 http://wordnet.princeton.edu

(synonym sets, o conjuntos de sinónimos) constituyen su unidad básica de significado. Cada uno de ellos se compone de un conjunto de las lexicalizaciones que representan un sentido y se identifica mediante un "offset" (byte) y su correspondiente PoS (Part of Speech); que puede ser (n) para nombres, (v) para verbos, (a) para adjetivos y (r) para adverbios. Por ejemplo:

02152053#n fish#1
01926311#v run#1
02545023#a funny#4
00005567#r automatically#1

El número que aparece después de cada palabra indica el número de sentido que representa el synset. Una palabra puede ser polisémica, esto es, tener varios significados. Por ejemplo:

02152053#n fish#1; representa la palabra fish con el sentido de pez como animal vertebrado acuático.

07775375#n fish#2; representa la palabra fish con el sentido de pescado como alimento.

También puede ser que varias palabras tengan el mismo significado, esto es, que sean sinónimas. En Wordnet están representados en el mismo synset:

02383458#n car#1, auto#1, automobile#1, machine#4, motorcar#1; representa el vehículo de cuatro ruedas.

Todos los synsets incluyen una glosa a modo de definición similar a la del diccionario tradicional, que describe el significado del concepto de forma explícita. Por ejemplo:

02383458#n car#1; 4-wheeled motor vehicle; usually propelled by an internal combustion engine: he needs a car to get to work;

Como ya se expuso anteriormente, WordNet está estructurada como una red semántica. Además de la información que pueden proporcionar los sinónimos (componentes del synset) y la glosa, tenemos que tener en cuenta los arcos de la red semántica, estos establecen diferentes relaciones entre los synsets, por ejemplo:

Hiperonimia: Es el término genérico usado para designar a una clase de instancias específicas. Y es un hiperónimo de X, si X es una clase de Y.

oak#n#2 HYPERONYM tree#n#1

Hiponimia: Es el término específico usado para designar el miembro de una clase, X es un hipónimo de Y, si X es una clase de Y. En el caso de los verbos se denomina Troponimia.

tree#n#1 HYPONYM oak#n#2

Antonimia: Es la relación que enlaza dos sentidos con significados opuestos.

active#a#1 ANTONYM_OF inactive#a#2
inactive#a#2 ANTONYM_OF active#a#2

Meronimia: Es la relación que se define como componente de, substancia de, o miembro de algo, X es merónimo de Y si X es parte de Y.

window#n#2 PART_OF car#n#1
protein#n#1 SUBSTANCE_OF milk#n#1
child#n#2 MEMBER_OF family#n#2

Holonimia: Es la relación contraria a la meronimia, Y es holónimo de X si X es una parte de Y.

car#n#1 HAS_PART window#n#2
milk#n#1 HAS_SUBSTANCE protein#n#1
family#n#1 HAS_MEMBER child#n#2

2.3.- EuroWordNet

El éxito de Wordnet impulsó la creación de nuevos proyectos similares para otros idiomas. De éstos el más destacado ha sido EuroWordNet[2] [Vossen, 1998]. EuroWordNet (EWN) consiste en una extensión multilingüe de WN, compuesta por bases de datos léxicas para 8 idiomas (inglés, holandés, castellano, italiano, francés, alemán, checo y estonio).

Además, siguiendo el modelo de EWN, empezaron a desarrollarse wordnets para el catalán, euskera, portugués, griego, búlgaro, ruso y sueco. Actualmente, la creación de estos wordnets esta coordinada por "Global Wordnet Organization"[3] y recientemente por la iniciativa "Open Multilingual WordNet"[4].

2.3.1.- El Indice InterLingüa

InterLingual Index (ILI, del inglés Índice InterLingüa) es un indice general de conceptos que permite conectar entre sí las unidades consideradas equivalentes en su significado en bases de datos de lenguas diferentes, permitiendo así pasar de términos de un idioma a términos de otro.

Cada wordnet local de EWN fue construido de forma independiente con los recursos disponibles en cada lengua, formando así un conjunto de módulos independientes. La conexión entre todos estos sistemas autónomos se hizo a través del Índice Interlingüal.

Cada índice en el ILI es un synset con una etiqueta de categoría morfosintáctica, una glosa y la referencia a su origen. Los synsets de cada wordnet particular están enlazados a algún índice del ILI. De esta forma, EWN proporciona la posibilidad de ir de una lexicalización de un concepto en una lengua determinada a otra lexicalización de ese mismo concepto en otra lengua diferente. Por ejemplo, partiendo del verbo español *convertirse* a través de su equivalente en el ILI, *to become*, se puede llegar a su correspondiente en italiano, *diventare*:

convertirse (esp) Eq_Near_Synonym to become(ILI) Eq_Near_Synonym (it) diventare

2 http://www.illc.uva.nl/EuroWordNet
3 http://www.globalwordnet.org
4 http://www.casta-net.jp/~kuribayashi/multi/

En la siguiente figura podemos ver cual es la arquitectura que usa EuroWordNet, y como se conectan los WordNets locales de diferentes idiomas a través del Índice InterLingüa.

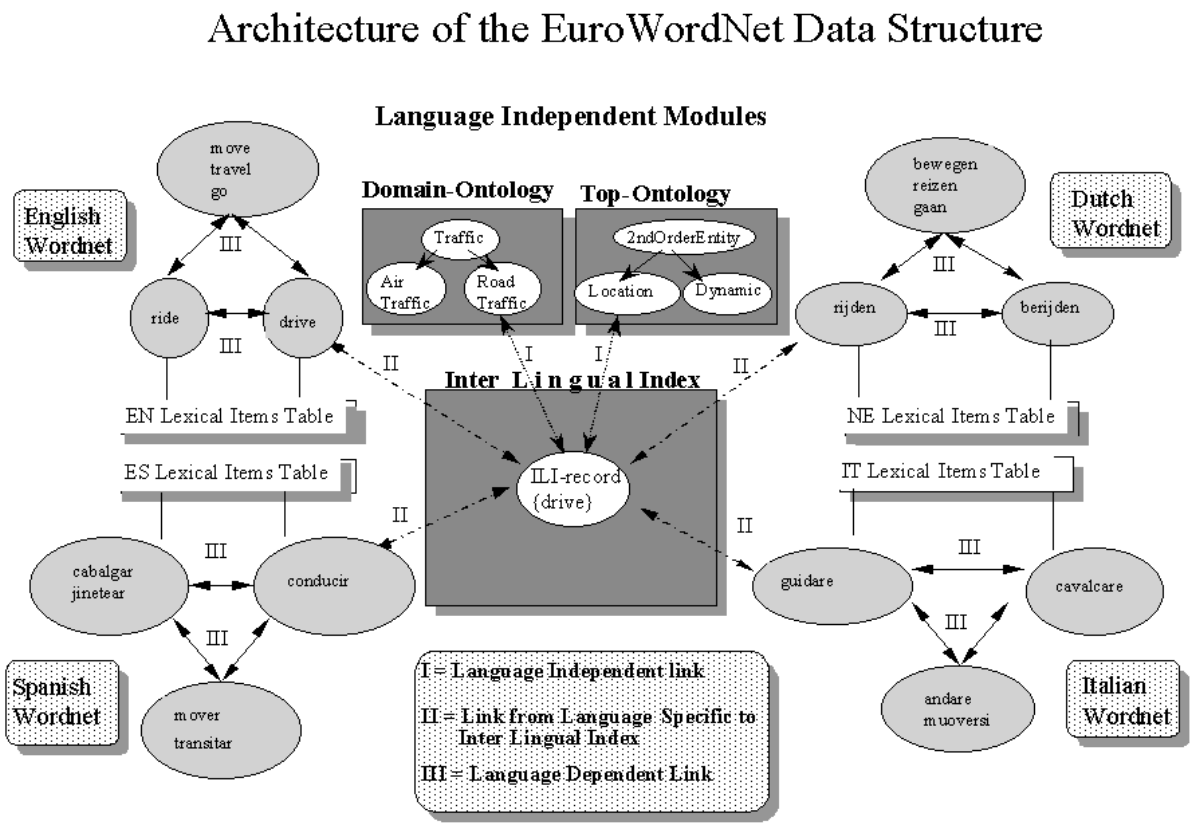


Imagen 2: Arquitectura de EuroWordNet

En la Imagen 2 se emplea el sentido *drive* para mostrar cómo se utilizan los diferentes tipos de enlaces de EWN. En este caso tenemos cuatro de los wordnets locales (ingles, holandés, italiano, castellano), para cada uno de ellos existe su correspondiente synset para el sentido que se está utilizando de ejemplo, en castellano sería *conducir*, en italiano *guidare*, en holandés *rijden* y en inglés *drive*. Estos synsets se relacionan con otros dentro de sus wordnets mediante enlaces dependientes del lenguaje (en la figura se identifican con el número romano III). Por ejemplo, en el caso del castellano, *conducir* estaría conectado mediante este tipo de enlaces con *transitar, cabalgar,* ... Por otra parte cada uno de ellos tiene un enlace desde su lenguaje específico a su correspondiente registro del Indice Interlingüa (número II). De este modo, al estar los cuatro synsets conectados al mismo registro ILI, podemos saber que todos representan el mismo concepto. Además, existe otro tipo de enlace independiente del lenguaje (número I) desde este

registro ILI a la información adicional que ofrecen las distintas ontologías sobre este sentido. Así, cada uno de los synsets de los cuatro wordnets del ejemplo, adquieren esas características representadas en ontologías asociadas al ILI, sin la necesidad de tener un enlace a las mismas.

2.4.- Multilingual Central Repository (MCR)

El Repositorio Central Multilingüe (MCR)[5], fue desarrollado como parte del proyecto MEANING [Rigau el al., 2002]. Su principal objetivo fue el de mantener la compatibilidad entre los diferentes wordnets y poder exportar, de forma consistente, el conocimiento adquirido para un idioma al resto de idiomas. Actualmente constituye probablemente una de las bases de conocimiento léxico multilingüe más extensa y más rica jamás construida.

El MCR (Multilingual Central Repository) [Atserias et al., 2004] es el resultado de la fusión de distintos recursos (diversas versiones de WordNet, ontologías y bases del conocimiento) que se llevó a cabo dentro del proyecto MEANING[6], KNOW[7] y KNOW2[8] y se sigue desarrollando en el proyecto SKATER[9]. Además mantiene el modelo utilizado en EWN, cuya arquitectura incluye InterLingual Index (ILI), WordNet Domains[10] y Top Ontology[11].

Su versión final esta integrada por wordnets para cinco idiomas diferentes (inglés, italiano, castellano, catalán y euskera) y contiene 1.642.384 relaciones semánticas únicas entre conceptos. También está enriquecido con 466.972 propiedades semánticas extraídas de otras fuentes, como WordNet Domains, Top Ontology o SUMO[12].

Para poder interactuar con el MCR se desarrolló WEI (Web Eurowordnet Interface) que es una interfaz web que permite hacer consultas y edición en el sistema. Se puede ver un ejemplo en la siguiente imagen.

5 http://adimen.si.ehu.es/web/MCR
6 http://www.lsi.upc.es/~nlp/meaning/demo/demo.html
7 http://ixa.si.ehu.es/know
8 http://ixa.si.ehu.es/know2
9 http://nlp.lsi.upc.edu/skater/
10 http://wndomains.itc.it/wordnetdomains.html
11 http://www.globalwordnet.org/gwa/ewn_to_bc/ewnTopOntology.htm
12 http://suo.ieee.org/SUO/SUMO/index.html

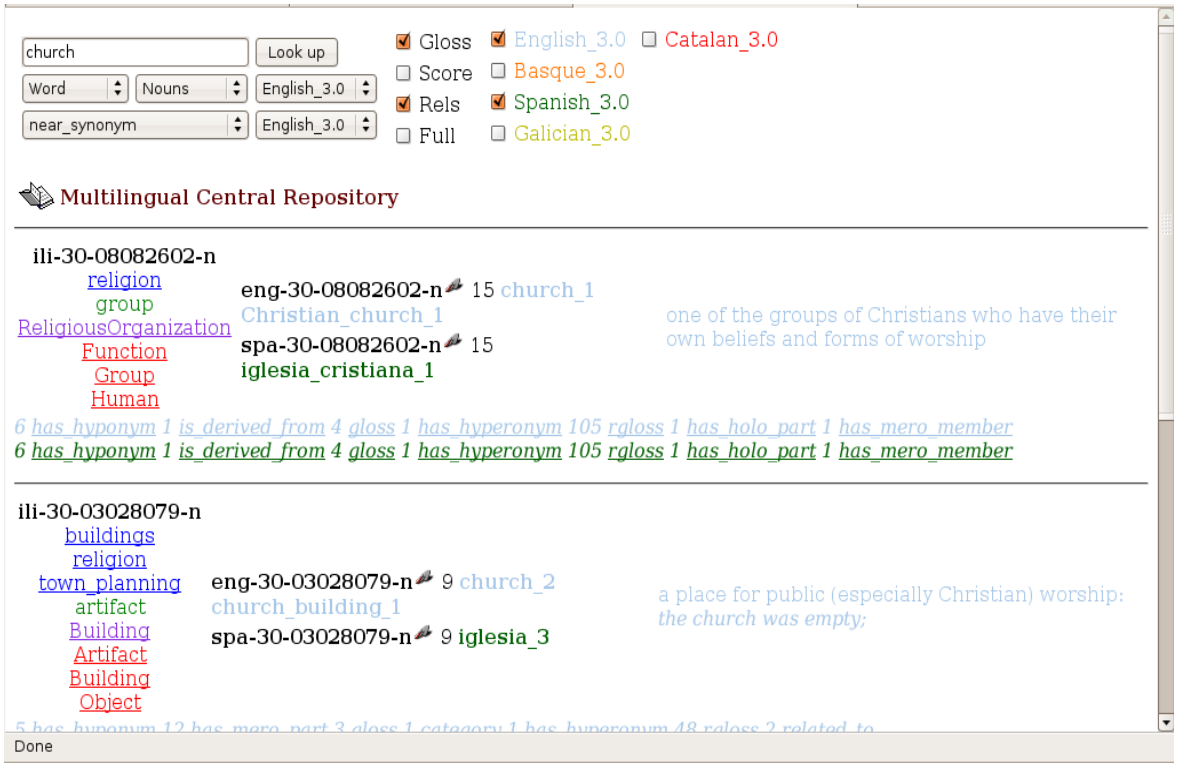


Imagen 3: Ejemplo de una consulta en WEI

Esta imagen representa una consulta sobre el wordnet English_3.0 de la palabra church. Los resultados corresponden a los sentidos de esa palabra en los wordnets English_3.0 (en azul) y Spanish_3.0 (en verde). A la izquierda de los sentidos aparece la información ontológica asignada a cada sentido. En este caso, como se han seleccionado las opciones Gloss y Rels, el sistema también ha devuelto una glosa (aparece a la derecha) y el número y tipo de relaciones que tiene cada sentido.

2.5- SSI-Dijkstra

SSI-Dijkstra [Cuadros & Rigau, 2007] es una versión del algoritmo SSI (Structural Semantic Interconnections) [Navigli & Velardi, 2005]. SSI es un algoritmo de WSD basado en el conocimiento (WSD: Word Sense Disambiguation) [Agirre & Edmonds, 2006]. El algoritmo SSI es muy simple y consiste en un paso de inicialización y en ciertos pasos iterativos (ver algoritmo debajo).

Dada W, una lista ordenada de palabras que deben ser desambiguadas, el algoritmo SSI-Dijkstra funciona de la siguiente manera: en el paso de inicialización, todas las palabras monosémicas son incluidas en una lista I que contiene las palabras que ya han sido interpretadas, mientras que las

palabras polisémicas son incluidas en la lista P, donde estarán todas las palabras pendientes de desambiguar. En cada paso, la lista I se usa para desambiguar una palabra de la lista P, seleccionando el sentido de la palabra más cercana a las palabras ya desambiguadas de la lista I. Una vez seleccionado el sentido, esa palabra la quitamos de la lista P y la incluimos en la lista I. El algoritmo termina cuando ya no quedan más palabras pendientes de desambiguar en la lista P.

```
SSI-Dijkstra (T: list of terms)
for each {t ∈ T} do
     I[t] = ∅
     if t is monosemous then
          I[t] := the only sense of t
     else
          P := P U {t}
     end if
end for
repeat
     P' := P
     for each {t ∈ P} do
          BestSense := ∅
          MaxValue := 0
          for each {sense s of t} do
         W[s] := 0
               N[s] := 0
         for each {sense s' ∈ I} do
              w := DijkstraShortestPath(s,s')
              if w > 0 then
                   W[s] := W[s] + (1/w)
             N[s] := N[s] + 1
              end if
                end for
                if N[s] > 0 then
                     NewValue := W[s]/N[s]
                     if NewValue > MaxValue then
                    MaxValue := NewValue
                    BestSense := s
               end if
                end if
          end for
          if MaxValue > 0 then
               I[t] := BestSense
         P := P \ {t}
          end if
     end for
until P ≠ P'
return (I,P);
```

Algoritmo 1: SSI-Dijkstra genérico

Inicialmente, la lista de palabras interpretadas I debería incluir los sentidos de las palabras monosémicas de W[13].

En relación a la proximidad de un sentido con el resto de sentidos de I, se usa el conocimiento de un grafo que contiene 99,635 nodos (synset) y 636,077 arcos. Este grafo incluye todas las relaciones sacadas WordNet y las glosas de WN[14]. Dijkstra es un algoritmo muy eficiente para calcular la distancia más corta entre dos nodos cualquiera del grafo [Cormen et al., 2001].

SSI-Dijkstra tiene unas propiedades muy interesantes. Por ejemplo, siempre que el grafo sea conexo, es capaz de calcular la distancia más corta entre dos nodos de cualquier grafo. Esto es, el algoritmo nos proporciona una respuesta.

Además, esta aproximación es independiente del idioma. El mismo grafo se puede usar para diferentes idiomas si existen palabras conectadas a WordNet para esa lengua.

2.6- SSI-Dijkstra+

SSI-Dijkstra+ [Laparra et al., 2010] es una versión extendida de SSI-Dijkstra. En ausencia de monosémicos, SSI-Dijkstra no proporciona ningún resultado, ya que depende directamente del conjunto I de palabras interpretadas para dar una solución. SSI-Dijkstra+ es una versión mejorada del algoritmo SSI-Dijkstra en dos aspectos.

En primer lugar, SSI-Dijkstra+ obtiene solución cuando los conjuntos de palabras por procesar no tienen monosémicos. Para ello, obtiene la palabra menos ambigua (la de menor grado de polisemia, o dicho de otro modo, la palabra con menos sentidos) y realiza una hipótesis acerca de su significado más correcto. La hipótesis se obtiene a partir del resultado de dos algoritmos diferentes: ASI o FSI.

ASI: Se basa en hallar la distancia mínima acumulada desde un sentido de la palabra hasta el resto de sentidos de las palabras de P. El sentido más probable de la palabra a tratar será aquel que tenga menor dicha distancia.

13 El algoritmo SSI-Dijkstra general no sabría qué hacer en ausencia de monosémicos. Para este caso particular, véase SSI-Dijkstra+, más adelante.

14 http://wordnet.princeton.edu/glosstag.shtml

FSI: Se basa en hallar la distancia mínima acumulada desde un sentido de la palabra hasta sólo el primer sentido de cada palabra de P. El sentido más probable de la palabra a tratar será aquel que tenga menor dicha distancia.

Estos dos algoritmos siempre obtienen solución aún cuando no hay monosémicos ya que aseguran que el conjunto de palabras interpretadas I no esté vacío antes del paso iterativo.

En segundo lugar, SSI-Dijkstra+ pretende mejorar la precisión de los resultados obtenidos a partir de la aplicación del algoritmo incluyendo para ello dos nuevos algoritmos durante el paso iterativo: ASP y FSP. Nótese que los dos algoritmos tienen en cuenta el conjunto de palabras interpretadas I para obtener su resultado. La diferencia entre ellos, se explica a continuación:

ASP: Trata de hallar la distancia mínima acumulada desde un sentido de la palabra por interpretar hasta cada uno de los sentidos presentes en I y el resto de sentidos de las palabras de P. El sentido más probable de la palabra a tratar será aquel que tenga menor dicha distancia.

FSP: Trata de hallar la distancia mínima acumulada desde un sentido de la palabra por interpretar hasta cada uno de los sentidos presentes en I y sólo el primer sentido de cada palabra de P. El sentido más probable de la palabra a tratar será aquel que tenga menor dicha distancia.

Podría ser también que por razones de precisión nos interesara no utilizar el conjunto P de palabras no interpretadas (podrían introducir error en algunos resultados aún no desambiguados), de la misma forma que incluir P podría dar resultados más precisos. La inserción de P o no a la hora de desambiguar palabras es una cuestión difícil que ha sido estudiada de forma empírica [Laparra et al., 2010]. Los resultados de dicho estudio fueron que los verbos se desambiguaban bastante mejor utilizando el FSP; mientras que el resto de palabras se desambiguaban con un resultado de precisión mejor cuando no se considera el conjunto de palabras sin desambiguar P.

En lo que respecta a este proyecto, se ha decidido utilizar FSI para la parte de inicialización (sólo si el conjunto I está vacío después de la inicialización inicial) y para la parte iterativa, se ha decidido usar FSP (sólo para verbos) o el propio algoritmo SSI-Dijkstra para el resto (nombres, adjetivos y adverbios).

2.8- UKB

UKB[15] [Agirre & Soroa, 2009] es un software para trabajar en el campo de la desambiguación de palabras. Codificado en C++, y apoyado en las librerías BoostGraph[16] se compone de clases para creación, lectura y procesamiento de grafos a partir de ficheros los cuales contienen la información necesaria para la desambiguación de palabras.

Teniendo en cuenta lo anterior, UKB ha sido uno de los pilares sobre los que se ha sustentado este proyecto debido a sus facilidades para trabajar con bases de conocimiento basadas en grafos. Por ejemplo, UKB proporciona facilidades para crear un fichero con la estructura del grafo sobre el que se va a trabajar, así como para procesar el fichero (lectura, búsqueda, inserción, etc.). UKB incluye diferentes ficheros con diferentes versiones del grafo de WordNet para trabajar con él. Así mismo, tiene también un diccionario para poder enlazar cada palabra a sus sentidos asociados (y las utilidades de uso de dicho diccionario).

Es importante también señalar que UKB proporcionaba casi todas las herramientas necesarias para este proyecto, pero no todas. Uno de los objetivos del proyecto ha sido intentar reutilizar al máximo el código propio de UKB, sin embargo hemos tenido que introducir utilidades y métodos que antes no tenía, que se explican a fondo un poco más adelante.

2.9- Historia del proyecto

Es importante destacar las diferentes fases por las que ha pasado el desarrollo de este proyecto.

La primera versión del algoritmo SSI-Dijkstra se codificó en Perl [Cuadros & Rigau, 2006], utilizando una librería que servía de interfaz con algunas de las funciones de BoostGraph. Con esto se consiguió una primera versión que, aunque incompleta era funcional. Pero el hecho de que hubiera que usar una interfaz Perl/C++ para usar las funciones BoostGraph era realmente complicado y de difícil mantenimiento.
Paralelamente, se estaba desarrollando UKB [Agirre & Soroa, 2009], que como ya se ha comentado utiliza directamente las librerías BoostGraph, y la versión 1.5 de UKB se acabó utilizando como pilar del primer proyecto de final de carrera desarrollado en relación a este proyecto [Blanco, 2010].

15 http://ixa2.si.ehu.es/ukb/
16 http://www.boost.org/doc/libs/1_42_0/libs/graph/doc/index.html

La primera versión de SSI-Dijkstra en C++ era una versión que ya usaba el software UKB como elemento principal. Sin embargo, esta primera versión no cubría todas las opciones de la versión anterior en Perl y había algunas cosas que podían hacerse mejor. De ahí surge el segundo proyecto relacionado con SSI-Dijkstra [Ledesma, 2012].

Este segundo proyecto completaba el primero, pero tampoco esta versión llegó a integrarse en UKB. Sin embargo, surgió la idea de mejorar aún más el algoritmo SSI-Dijkstra, dando cabida al algoritmo SSI-Dijkstra-Fast. Debido a esto, surgió el tercer proyecto relacionado con esta familia de algoritmos [Chihuailaf, 2013], un proyecto desarrollado en la UTEM (Universidad Tecnológica Metropolitana) de Chile y que implementa una versión preliminar del algoritmo para UKB 1.6.

Este tercer proyecto, implementaba todo lo que en ese momento se necesitaba relacionado con la familia de algoritmos SSI-Dijkstra. Pero, este tercer proyecto tenía dos problemas: estaba codificado para la versión 1.6 de UKB e implementaba el algoritmo de una forma compleja e ineficiente.

En paralelo, UKB lanzaba a su versión 2.0, realizando cambios importantes en la estructura nuclear de la herramienta, consiguiendo así trabajar con estructuras mucho más grandes de los que la versión anterior permite.

El presente proyecto, bebe básicamente de las dos fuentes anteriores: el último (y tercer) proyecto relacionado con SSI-Dijkstra, y la versión 2.0 de UKB; para unificar ambos. Además de ello trata de realizar una versión de la familia de algoritmos de SSI-Dijkstra que pueda ser más adelante lanzada al público como una funcionalidad más de UKB.

3.- DOP: Documento de Objetivos del Proyecto

3.1.- Objetivos

Este proyecto surge de la necesidad de disponer del algoritmo SSI-Dijkstra-Fast en C++ y compatible con la última versión de UKB 2.0. SSI-Dijkstra-Fast es un algoritmo de desambiguación de palabras. Existe una versión del algoritmo que usa una versión anterior de UKB (1.6) en lugar de la versión que actualmente existe (2.0). Se pretende crear un nuevo algoritmo compatible con la versión 2.0 de UKB para generar y realizar consultas sobre grafos más grandes de lo que permite la versión 1.6 de UKB.

Se quiere también recodificar la parte del algoritmo SSI-Dijkstra-Fast para hacerla más legible y más eficiente que la del algoritmo existente, en búsqueda de una mejora temporal que en principio el algoritmo ya tiene frente a su predecesor, SSI-Dijkstra+.

También se quiere probar esta nueva versión y demostrar empíricamente que funciona igual que las versiones anteriores de SSI-Dijkstra, pero más rápido.

Además, se pretende aprovechar el software de desambiguación de palabras basadas en grafos explicado anteriormente (UKB) y el cual ya tiene codificados varios métodos de desambiguación, basados en PageRank[17] pero no todos los relacionados con SSI-Dijkstra en concreto.

Por tanto, el objetivo del proyecto es doble: por una parte, se intenta crear el algoritmo SSI-Dijkstra-Fast compatible con la última versión del software UKB y por otra parte se intenta añadir la familia de algoritmos SSI-Dijkstra al software de UKB para poder usarlo en tareas de desambiguación de palabras.

3.2.- Método de trabajo

Durante el proyecto, el método de trabajo será el siguiente: se trabajará por prototipos (cada iteración finalizada contará con un pequeño prototipo que da funcionalidad a una parte de la aplicación). El proyecto se desarrollará de forma iterativa e incremental, los prototipos de cada

17 http://es.wikipedia.org/wiki/PageRank

iteración añadirán funcionalidades al prototipo de la iteración anterior.

El profesor irá marcando objetivos que el alumno debe solventar en un lapso de tiempo acordado entre ambos. El alumno dispone de medios (E-mail del profesor, documentación específica, Internet, etc.) para solventar sus dudas y en caso de no poder llevar la tarea a tiempo se pospondrá hasta que el alumno lo tenga dispuesto o en su defecto, se avisará al profesor y se acordará un día para realizar una reunión y poder así solventar dudas y avanzar en el estado el proyecto.

Las reuniones serán un punto para elegir nuevos objetivos y observar la evolución de los realizados (o no) hasta el momento, por lo que es importante que el alumno lleve todo el material posible en relación a su proyecto. Si alguno de los objetivos planteados no está correctamente realizado, el alumno tiene la responsabilidad de intentar completarlo de forma adecuada para una próxima ocasión, así como el profesor intentará guiar al alumno en este hecho. Según se vaya progresando en la realización del proyecto, los objetivos irán acercándose cada vez más a la finalización del mismo, y así hasta el final.

El proyecto se dividirá en dos iteraciones (re-escritura de los algoritmos SSI-Dijkstra y SSI-Dijkstra+ para hacerlos compatibles con UKB 2.0 y creación del algoritmo SSI-Dijkstra-Fast), las cuales a su vez se subdividirán en cinco tareas más: Captura de Requisitos, Análisis, Diseño, Implementación y Pruebas.

En caso de estancamiento o problemas relacionados con el proyecto, se seguirá el plan de contingencia.

3.3.- Elección tecnológica

Dadas las especificaciones del proyecto, es necesario trabajar con unas herramientas y tecnologías predeterminadas. Sin embargo, se han podido hacer algunas elecciones importantes.

3.3.1.- Pre-requisitos

Antes de pasar a comentar las decisiones acerca de la tecnología utilizada para el proyecto, es necesario comentar alguno de los pre-requisitos para poder utilizar y/o desarrollar con este

software.

En primer lugar, es importante que la máquina que vayamos a utilizar sea capaz de compilar C++, ya que el proyecto completo está hecho en C++. Este es un pre-requisito bastante claro a tener en cuenta.

También es importante saber que el proyecto no tiene sentido sin estar integrado a UKB, ya que utiliza las clases y métodos proporcionadas por este software.

Es necesario tener en cuenta también que tanto UKB como el proyecto utilizarán librerías BoostGraph de C++.

Por último, es importante que la máquina en la que se va a trabajar tenga instalada e integrada alguna versión de Git (necesario para trabajar con Github, que es donde está el proyecto de UKB principal y donde al final se integrará el proyecto a UKB).

Los requisitos indispensables respecto a elección tecnológica es el uso de C++, UKB, BoostGraph y Git.

3.3.2.- Desarrollo: la máquina personal del alumno

3.3.2.1.- Sistema operativo

El sistema operativo elegido para el desarrollo del proyecto será Linux Ubuntu 13.04 (Raring Ringtail). Se ha elegido un SO Linux frente a cualquier otra opción debido a las facilidades que el propio sistema ofrece para desarrollar software en C++. De igual forma, se ha elegido Ubuntu debido a que el alumno ya estaba familiarizado de algún modo con sistemas operativos de familia Debian y claramente, era más fácil trabajar en él (sin tener que aprender las particularidades de un sistema operativo diferente).

3.3.2.2.- Entorno de desarrollo

Para el desarrollo del proyecto se utilizará la versión 7.0.1 de Netbeans. Se usará este entorno de

desarrollo ya que el alumno lo conoce y resultaba cómodo trabajar con él con algunas de las herramientas con las que hay que trabajar (por ejemplo, Git).

Este entorno de desarrollo también nos permite modificar aspectos del *coding standart* y por tanto es adecuado para trabajar en este proyecto.

3.3.2.3.- VMWare Player

Es un producto gratuito que permite ejecutar máquinas virtuales creadas con productos de VMWare. Las máquinas virtuales se pueden crear con productos más avanzados como VMWare Workstation, o con el propio VMWare Player desde su versión 3.0 (las versiones anteriores no incluyen dicha funcionalidad).

3.3.3.- Documentación

Como no todo es desarrollo de software en un proyecto de final de carrera, se deja también una relación del software utilizado para tareas no relacionadas directamente con la implementación, pero sí con el proyecto.

Respecto al paquete ofimático utilizado, hay que señalar que en ocasiones el alumno trabajaba también en su memoria en un sistema operativo Windows, por lo que se ha preferido el paquete ofimático OpenOffice frente al LibreOffice (a pesar de que este último viene instalado por defecto en la versión de Ubuntu con la que el alumno trabaja).

3.3.3.1.- OpenOffice Writer. Memoria

OpenOffice.org Writer es un procesador de texto multiplataforma que forma parte del conjunto de aplicaciones de la suite ofimática OpenOffice.org. Además de otros formatos estándares, puede abrir y grabar el formato propietario .doc de Microsoft Word casi en su totalidad. El formato nativo para exportar documentos es XML. También puede exportar a ficheros PDF nativamente sin usar programas intermedios.

3.3.3.2.- OpenOffice Impress. Transparencias

OpenOffice.org Impress es un programa de presentación similar a Microsoft PowerPoint. Es parte de la suite de oficina de OpenOffice.org desarrollada por Sun Microsystems. Incluye la capacidad de crear archivos PDF. Impress sufre de la carencia de diseños de presentación listos para usarse. Sin embargo, se pueden obtener fácilmente en Internet plantillas de terceros.

3.3.3.3.- Gantt Project. Diagrama de Gantt

Gantt Project es una herramienta de escritorio multiplataforma para la planificación y gestión de proyectos. Funciona en Windows, Linux y MacOSX y es gratis y de código abierto. Fundamentalmente, se utiliza para hacer diagramas de Gantt aunque permite hace también diagramas PERT y tablas de asignación de recursos, además de otras funcionalidades. Permite además exportar sus diagramas en formato de imagen, lo cual será bastante interesante a la hora de integrarlos en esta memoria.

3.3.3.4.- Visual Paradigm for UML 10.2. Diagrama UML

Visual Paradigm for UML (VP-UML) es una herramienta de diseño UML y herramienta CASE UML diseñado para ayudar al desarrollo de software. VP-UMLsoporta los estándares de modelado clave como Unified Modeling Language (UML) 2.4, SoaML, SysML, ERD, DFD, BPMN 2.0, 2.0 ArchiMate, etc. Es compatible con los equipos de desarrollo de software de captura de requerimientos, planificación de software (utilizar el análisis de casos), ingeniería de código, modelado de clase, el modelado de datos, etc.

3.4.- Alcance

3.4.1.- Recursos humanos

Los recursos humanos con los que cuenta este proyecto son el alumno y el profesor asignado como director de proyecto así como Aitor Soroa (uno de los desarrolladores de UKB) quien también podrá ayudar como consultor en algunos aspectos del proyecto. Es responsabilidad del alumno seguir las

indicaciones del profesor respecto a objetivos, tareas y plazos. Es responsabilidad del profesor orientar al alumno en la ejecución de los objetivos para alcanzar el éxito lo antes posible.

3.4.2.- Recursos materiales

El alumno dispone de un ordenador personal portátil con un sistema operativo Linux instalado (en concreto, una distribución Ubuntu) para el desarrollo del proyecto. El alumno dispone también de material de oficina (véase como: papel, bolígrafos, impresora, etc) para obtener e imprimir documentación, realizar esquemas, croquis, etc. Se dispone también de una unidad de disco extraíble USB para realizar copias de seguridad, así como un disco duro externo en el que se puede realizar una segunda copia de seguridad en caso de necesitarlo.

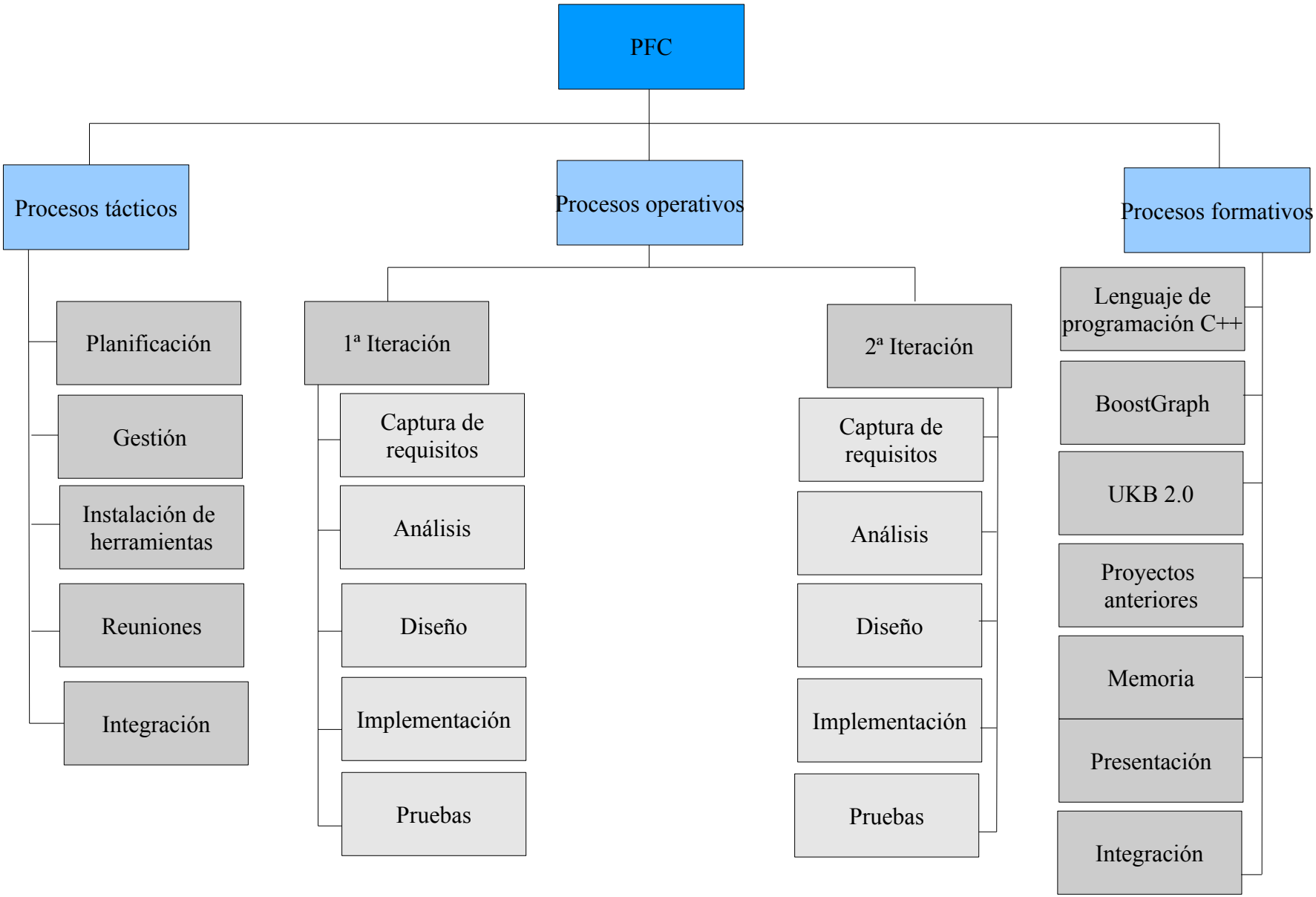
El profesor dispone de un ordenador de sobremesa con acceso a Internet, y del mencionado anteriormente material de oficina, por lo que se podrán hacer esquemas, dibujos, etc. en caso de ser necesario.

3.4.3.- Recurso económico

Al ser un proyecto de final de carrera, y no haber recurso económico de por medio, el coste total del proyecto será nulo, presuponiendo que los partícipes del proyecto tenían ya todo el material independientemente y anteriormente a este proyecto. De todas formas, casi todo el software utilizado para desarrollar el proyecto, así como para escribir la documentación y la memoria es código abierto (a excepción de VMWare Player, que es gratis pero no abierto), y, más importante para este apartado, gratis. Por lo que en cualquier caso, el alumno no ha tenido que gastar dinero en licencias de ningún tipo ya que el software utilizado no es ni privativo ni cuesta dinero.

3.4.4.- Diagrama EDT (Estructura de descomposición del trabajo)

En la siguiente hoja se adjunta el diagrama correspondiente a la Estructura de Descomposición del Trabajo planificada para este proyecto. En la siguiente, hay un desglose de las tareas, así como una pequeña descripción de cada una de ellas y una referencia temporal que indica cuánto tiempo estimado transcurrirá en cada una de las tareas.


PFC
Procesos tácticos
Procesos operativos
Procesos formativos
Planificación
Gestión
Instalación de herramientas
Reuniones
Integración
1ª Iteración
Captura de requisitos
Análisis
Diseño
Implementación
Pruebas
2ª Iteración
Captura de requisitos
Análisis
Diseño
Implementación
Pruebas
Lenguaje de programación C++
BoostGraph
UKB 2.0
Proyectos anteriores
Memoria
Presentación
Integración

3.4.5.- Proyecto de Final de Carrera (PFC)

Procesos formativos

1. Lenguaje de programación: C++

Descripción: Recordatorio breve de programación básica en C++. Esto es necesario debido a que el alumno no es muy familiar con C++, y a pesar de haberlo utilizado en un proyecto anterior no es un lenguaje que habitualmente use.
Tiempo estimado: 5 horas

2. BoostGraph

Descripción: Recordatorio breve de utilización de la librería BoostGraph en C++ para el desarrollo del proyecto.
Tiempo estimado: 5 horas

3. UKB 2.0

Descripción: Revisión del código y la documentación disponible de UKB 2.0 para conseguir entender cómo utilizar este software y qué diferencias se encuentran con la versión 1.6 de UKB.
Tiempo estimado: 15 horas

4. Proyectos anteriores

Descripción: Revisión de los proyectos anteriormente realizados por otros alumnos relacionados con el proyecto con el objetivo de entender y mejorar las versiones realizadas por ellos.
Tiempo estimado: 20 horas

5. Memoria

Descripción: Realización de la memoria con el objetivo de recopilar todos los pasos dados durante el proyecto.
Tiempo estimado: 60 horas

6. Presentación

Descripción: Realización de las transparencias y preparación de la presentación de finalización del proyecto.

Tiempo estimado: 10 horas.

7. Integración

Descripción: Revisión y estudio herramientas y métodos de integración (en concreto los los relacionados con Git y Github).

Tiempo estimado: 10 horas.

Total procesos formativos: 125 horas.

Procesos tácticos

1. Planificación

Descripción: Realización de las tareas de planificación (y re-planificación en su caso) del proyecto, entre otras, el propio DOP.

Tiempo estimado: 25 horas

2. Gestión

Descripción: Realización de las tareas de gestión del proyecto.

Tiempo estimado: 15 horas

3. Instalación de las herramientas

Descripción: Instalación de las herramientas (tácticas, por ejemplo: Gantt Project; u operativas, por ejemplo: UKB).

Tiempo estimado: 10 horas

4. Reuniones

Descripción: Reuniones con el director de proyecto, sea para tratar sobre temas tácticos (planificación y gestión, memoria, etc) u operativos (el propio código del proyecto).

Tiempo estimado: 30 horas

5. Integración

Descripción: Tareas de integración de la familia de algoritmos SSI-Dijkstra en UKB.

Tiempo estimado: 10 horas

Total procesos tácticos: 90 horas.

Procesos operativos

1. Primera iteración

Descripción: Esta primera iteración cubre todos los aspectos operativos desde el inicio del proyecto hasta que el algoritmo del SSI-Dijkstra+ existente en la versión del proyecto de Aritza tenga un equivalente funcional en la versión 2.0 de UKB.

(a) Captura de requisitos

Descripción: Captura de requisitos de la primera iteración.

Tiempo estimado: 3 horas

(b) Análisis

Descripción: Análisis de la primera iteración

Tiempo estimado: 8 horas.

(c) Diseño

Descripción: Diseño de la primera iteración.

Tiempo estimado: 8 horas.

(d) Implementación

Descripción: Implementación de la primer iteración.

Tiempo estimado: 12 horas.

(e) Pruebas

Descripción: Pruebas de la primera iteración.

Tiempo estimado: 5 horas.

Total primera iteración: 36 horas.

2. Segunda iteración

Descripción: Esta segunda iteración cubre todos los aspectos operativos desde la finalización de la primera iteración hasta la hasta la finalización del proyecto (codificación de SSI-Dijkstra-Fast compatible con la versión 2.0 de UKB).

(a) Captura de requisitos

Descripción: Captura de requisitos de la segunda iteración.

Tiempo estimado: 6 horas.

(b) Análisis

Descripción: Análisis de la segunda iteración.

Tiempo estimado: 8 horas

(c) Diseño

Descripción: Diseño de la segunda iteración.

Tiempo estimado: 5 horas.

(d) Implementación

Descripción: Implementación de la segunda iteración.

Tiempo estimado: 15 horas.

(e) Pruebas

Descripción: Pruebas de la segunda iteración.

Tiempo estimado: 10 horas.

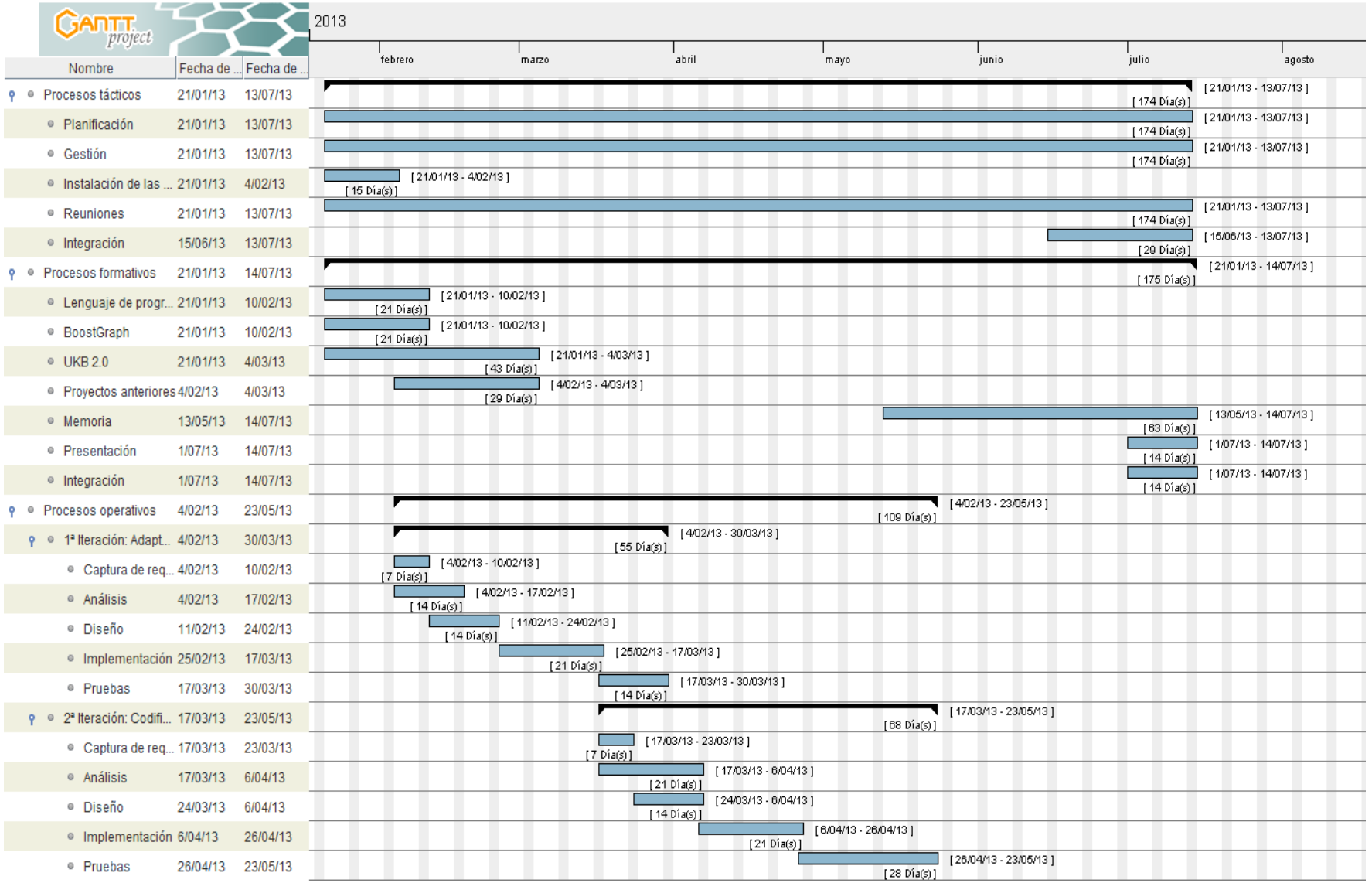
Total segunda iteración: 44 horas.

Total procesos operativos: 80 horas.

Total tiempo estimado PFC: 295 horas.

3.5.- Planificación temporal: Diagrama de Gantt

En la siguiente hoja, se adjunta el diagrama de Gantt de tareas planificadas para este proyecto. Junto a cada una de las barras de las tareas, a la derecha, puede verse el día de inicio y el día final estimado de cada una de las tareas; así como la duración total de cada una de ellas, lo cual puede verse debajo.


GANTT project
2013
febrero
marzo
abril
mayo
junio
julio
agosto
Nombre
Fecha de ...
Fecha de ...
Procesos tácticos 21/01/13 13/07/13 [21/01/13 - 13/07/13] [174 Día(s)]
Planificación 21/01/13 13/07/13 [21/01/13 - 13/07/13] [174 Día(s)]
Gestión 21/01/13 13/07/13 [21/01/13 - 13/07/13] [174 Día(s)]
Instalación de las ... 21/01/13 4/02/13 [21/01/13 - 4/02/13] [15 Día(s)]
Reuniones 21/01/13 13/07/13 [21/01/13 - 13/07/13] [174 Día(s)]
Integración 15/06/13 13/07/13 [15/06/13 - 13/07/13] [29 Día(s)]
Procesos formativos 21/01/13 14/07/13 [21/01/13 - 14/07/13] [175 Día(s)]
Lenguaje de progr... 21/01/13 10/02/13 [21/01/13 - 10/02/13] [21 Día(s)]
BoostGraph 21/01/13 10/02/13 [21/01/13 - 10/02/13] [21 Día(s)]
UKB 2.0 21/01/13 4/03/13 [21/01/13 - 4/03/13] [43 Día(s)]
Proyectos anteriores 4/02/13 4/03/13 [4/02/13 - 4/03/13] [29 Día(s)]
Memoria 13/05/13 14/07/13 [13/05/13 - 14/07/13] [63 Día(s)]
Presentación 1/07/13 14/07/13 [1/07/13 - 14/07/13] [14 Día(s)]
Integración 1/07/13 14/07/13 [1/07/13 - 14/07/13] [14 Día(s)]
Procesos operativos 4/02/13 23/05/13 [4/02/13 - 23/05/13] [109 Día(s)]
1ª Iteración: Adapt... 4/02/13 30/03/13 [4/02/13 - 30/03/13] [55 Día(s)]
Captura de req... 4/02/13 10/02/13 [4/02/13 - 10/02/13] [7 Día(s)]
Análisis 4/02/13 17/02/13 [4/02/13 - 17/02/13] [14 Día(s)]
Diseño 11/02/13 24/02/13 [11/02/13 - 24/02/13] [14 Día(s)]
Implementación 25/02/13 17/03/13 [25/02/13 - 17/03/13] [21 Día(s)]
Pruebas 17/03/13 30/03/13 [17/03/13 - 30/03/13] [14 Día(s)]
2ª Iteración: Codifi... 17/03/13 23/05/13 [17/03/13 - 23/05/13] [68 Día(s)]
Captura de req... 17/03/13 23/03/13 [17/03/13 - 23/03/13] [7 Día(s)]
Análisis 17/03/13 6/04/13 [17/03/13 - 6/04/13] [21 Día(s)]
Diseño 24/03/13 6/04/13 [24/03/13 - 6/04/13] [14 Día(s)]
Implementación 6/04/13 26/04/13 [6/04/13 - 26/04/13] [21 Día(s)]
Pruebas 26/04/13 23/05/13 [26/04/13 - 23/05/13] [28 Día(s)]

3.6.- Plan de contingencia

En la siguiente tabla se presenta la relación de riesgos posibles durante la ejecución de las tareas del proyecto. El plan de contingencia a seguir en cada uno de los casos, viene descrito debajo.

Riesgo	Probabilidad	Gravedad
Dificultad en entender	Medio	Bajo
Pérdida de la documentación	Muy bajo	Bajo
Falta de disponibilidad para realizar reuniones	Bajo	Bajo
Retraso en la tarea	Medio	Medio
Ordenador estropeado	Medio	Bajo
Pérdida de datos	Bajo	Medio
Problemas en manejar software	Bajo	Medio

Tabla 1: Tabla de riesgos

Dificultad en entender

Riesgo: Dificultad del alumno para entender algo relacionado con el proyecto que le haga perder excesivo tiempo, con lo que le llevaría al riesgo de no poder realizar la tarea a tiempo.

Solución: El alumno le informará al profesor encargado del proyecto del problema, y acordarán realizar alguna reunión para aclarar las dudas o le informara dónde puede encontrar información añadida para solucionarlo o ampliaran el plazo de la tarea para que tenga más tiempo para hacerlo.

Pérdida de la documentación

Riesgo: El alumno pierde documentación o acceso a la documentación que dispone para la formación que dispone.

Solución: La mayoría de la documentación del alumno es física (papel escrito). En cualquier caso, hay un "backup" de sitios de donde se ha obtenido dicha información en el correo del alumno, ya que el profesor le envía un correo electrónico por reunión con los aspectos más relevantes de la reunión, incluyendo enlaces a documentación interesante.

Falta de disponibilidad para realizar reuniones

Riesgo: El alumno o el profesor encargado no pueden acudir a una reunión concertada por causas tales como enfermedad, causas personales, trabajo...

Solución: Se notificará dicha indisponibilidad a la otra parte, y en su caso, se acordará otro horario (fecha y hora) para la reunión.

Retraso en la tarea

Riesgo: El alumno no ha conseguido realizar la tarea a tiempo.

Solución: Entre el alumno y el profesor acuerdan un nuevo plazo para la ejecución de la tarea, así como posponen nuevos objetivos.

Ordenador estropeado

Riesgo: El ordenador del alumno sufre daños parciales o totales

Solución: El profesor y el alumno llegarán a un acuerdo. El profesor puede prestarle un ordenador de forma temporal, así como el alumno, puede usar los ordenadores de uso público de la facultad de Informática.

Pérdida de datos

Riesgo: Se pierden los datos guardados en el ordenador.

Solución: Se recupera el último "backup" de la memoria extraíble USB del alumno y se intenta devolver el proyecto al estado avanzado anterior.

Problemas en manejar software

Riesgo: Problemas en manejar software que está utilizando el alumno, debido a desconocimiento del mismo.

Solución: Intentar buscar toda la documentación posible por Internet, por los libros de la biblioteca, manuales... El profesor encargado le podrá orientar al alumno sobre que material le convendría.

3.7.- Análisis de factibilidad

Una vez conocidas las tareas a realizar para la finalización del proyecto y el tiempo estimado en cada una de ellas, también conocido la posibilidad y descripción de los riesgos de las tareas y sus posibles soluciones; creemos que el proyecto parece factible en el intervalo de tiempo dado.

4.- Captura de requisitos

4.1.- Motivación

Como ya hemos dicho, este proyecto surge de la necesidad de disponer de la familia de algoritmos SSI-Dijkstra codificados en C++ y compatibles con la versión actual de UKB. Existen versiones de los algoritmos válidos para la versión 1.6 de UKB, pero son incompatibles con la versión 2.0. Además, se pretende revisar y mejorar la eficiencia temporal del algoritmo SSI-Dijkstra-Fast ya existente debido a que se cree que en dicho algoritmo usa una estructura de datos ineficiente y existen pasos innecesarios, los cuales podrían evitarse.

En paralelo, se trabaja con la herramienta ya mencionada UKB, la cual está escrita en C++ y utiliza también las librerías BoostGraph. UKB es una herramienta que permite la desambiguación de palabras mediante grafos (como SSI-Dijkstra), pero que no implementa ningún algoritmo de la familia SSI-Dijkstra.

Se pensó en aprovechar ambas cosas para generar un algoritmo en C++ que trabajara con UKB para, dar un mejor servicio en cuanto al tiempo y calidad de respuesta desambiguando palabras y de paso conseguir que UKB trabajara con SSI-Dijkstra también. Por lo que, era también requisito importante del proyecto que SSI-Dijkstra fuera capaz de entender, procesar y trabajar con la misma entrada que UKB y que SSI-Dijkstra fuera capaz de dar una salida de formato idéntica a la de UKB (por las aplicaciones que puedan trabajar con la salida de UKB para obtención de otros datos...).

4.2.- Requisitos específicos

Además de todo eso, es necesario señalar que este proyecto, debe ser compatible también con la funcionalidades que ya nos ofrecían las versiones compatibles con la versión 1.6 de UKB. Las funcionalidades son básicamente las siguientes:

1. El algoritmo debe ser capaz de utilizar los tres tipos de ordenación ya definidos para proyectos anteriores (*poly:* ordenación por polisemia, *expl:* ordenación explícita y sin ordenación)
2. El algoritmo debe dar la opción al usuario de trabajar con optimización o sin optimización al

igual que en los anteriores proyectos.

3. El algoritmo debe ser capaz de trabajar con los grafos generados con la herramienta *compile_kb* de UKB[18].
4. El algoritmo debe de ser capaz de generar una salida evaluable por el *scorer2* de Senseval[19], y debe tener en cuenta evaluar un tipo de concepto concreto de una manera diferente a cómo lo hace UKB.

4.3.- Resumen

En resumen, los requisitos explícitos del proyecto son:

1. Recodificar los algoritmos SSI-Dijkstra y SSI-Dijkstra+ en C++ (haciéndolos compatibles con la versión 2.0 de UKB).
2. Codificar una nueva versión del algoritmo SSI-Dijkstra-Fast utilizando las librerías BoostGraph y el software UKB para ello.
3. Conseguir que este proyecto cumpla también las especificaciones relacionadas con proyectos anteriores.

18 Aunque se profundizará en esto más adelante, debido al cambio de versión del software UKB, los grafos generados con la herramienta *compile_kb* sin más no son compatibles con este proyecto (hace falta utilizar una herramienta de conversión incluida en UKB para poder usarlos).

19 Se detallará esto algo más adelante, en el apartado de pruebas.

5.- Análisis

Se puede observar que este proyecto tiene bastante poco peso en cuanto a análisis, pero sí hay algunas cosas que podrían comentarse antes de seguir a otras cuestiones.

5.1.- Sobre UKB

Como se ha comentado anteriormente, UKB ha sido un software fundamental para el proyecto, pero que a pesar de todo no cumplía del todo con las necesidades demandadas por la aplicación desarrollada. Sin embargo, para tratar de modificar lo menos posible el código de UKB, se han creado elementos nuevos que interactúan con los algoritmos del proyecto en lugar de los de UKB. A continuación se explica un poco por encima con qué herramientas contábamos y qué necesitábamos, en el apartado de diseño se puede ver un diagrama UML de cómo se encuentra la herramienta actualmente y en el apartado de implementación se detalla más en profundidad cuáles han sido concretamente los cambios realizados en la herramienta.

UKB es una herramienta que como se ha comentado en proyectos anteriores [Blanco, 2010] cuenta con todas las estructuras de datos que la familia de algoritmos SSI-Dijkstra necesita para poder funcionar. La diferencia fundamental entre la versión 1.6 y 2.0 es precisamente que ahora la aplicación es capaz de funcionar con grafos más grandes (aunque ahora una vez creado el grafo es completamente imposible realizar ciertas operaciones en él, al menos de manera directa; lo cual ha introducido cambios en cómo SSI-Dijkstra realiza algunos de sus cálculos); pero eso no ha variado las estructuras de datos nucleares de UKB.

Dado que SSI-Dijkstra necesita trabajar con un tipo concreto de concepto que en UKB se procesa de manera diferente a la que esperaríamos en SSI-Dijkstra, y que SSI-Dijkstra genera a veces una salida diferente a la que genera UKB (el usuario debería poder decidir si quiere los identificadores de los contextos o no en la salida, entre otras cosas), se ha visto necesario crear nuevas clases (que, a pesar de todo, heredan de las clases de UKB) para evitar colisiones entre los programas que usan las clases de UKB como núcleo y la familia de algoritmos SSI-Dijkstra.

Teniendo en cuenta los requisitos que debía de cumplir la aplicación a desarrollar, se tomaron las siguientes decisiones:

- Usar la función de lectura de datos (del fichero de contextos) de UKB, pero no la función de salida de datos (se definiría una propia). Uno de los requisitos del proyecto es mostrar los datos de una forma muy concreta (pudiendo, por ejemplo no mostrar el contexto al que pertenecen las palabras o bien, no mostrar palabras que se daban ya desambiguadas en el propio contexto).
- Usar para compilar los grafos el propio programa *compile_kb* proporcionado por UKB, y utilizar también el programa *convert2.0* para convertir esos grafos compilados en un grafo compatible con la versión 2.0 de UKB. Esto facilita también el tratamiento del grafo, ya que todas las funciones de búsqueda, lectura y tratamiento del grafo están ya implementadas para UKB.

5.2.- Sobre SSI-Dijkstra

Hay cosas a comentar que claramente recaen sobre el propio SSI-Dijkstra en sí ya que son cosas que tendremos en cuenta en la implementación final del algoritmo:

- Para capturar las diferentes posibilidades en la llamada al programa que se generará, se usará el paquete de librerías program_options, que ya vienen integrados en BoostGraph y que facilitan bastante utilizar diferentes opciones.
- Los métodos de ordenación se implementarán usando las opciones proporcionadas por el propio lenguaje (el método *sort* de C++), en lugar de definir una función propia para ello. Además de ser más cómodo para el programador, será más eficiente.

5.3- SSI-Dijkstra-Fast

SSI-Dijkstra-Fast nace como una idea para tratar de hacer más eficiente SSI-Dijkstra+. Hasta ahora, todos los algoritmos de la familia SSI-Dijkstra han trabajado de la misma manera: a partir de cada palabra polisémica obtenemos la distancia mínima acumulada al conjunto de monosémicas y nos quedamos con el sentido de menor valor; de forma que calculamos tantos Dijkstras como sentidos tenemos en el conjunto de palabras polisémicas.

En lugar de eso, este nuevo algoritmo trabaja en sentido contrario: desde cada palabra del conjunto de monosémicas (o desambiguadas), calculamos un Dijkstra para obtener la distancia a cada uno de los demás sentidos del grafo (es importante recordar que el algoritmo de Dijkstra

obtiene la información asociada a un nodo origen para todos los demás nodos del grafo), guardando esta información en una estructura de datos adecuada para así poder obtener la distancia desde cada sentido del conjunto de polisémicas (o sin desambiguar) sin necesidad de recorrer de nuevo el grafo (buscando esa distancia en la estructura de datos ya creada).

Teóricamente, este cambio permitiría al algoritmo una mejora del rendimiento temporal ya que el orden de ejecución del algoritmo no depende de dos factores (número de palabras polisémicas y número medio de sentidos por palabra), sino que dependería de uno sólo (el número total de palabras), por lo que sería un orden de ejecución cercano a orden lineal. Forma parte de los objetivos de este proyecto, demostrar empíricamente que efectivamente este algoritmo es mejor en el aspecto temporal que sus predecesores.

```
SSIDF (T: list of terms)
for each {t ∈ T} do
     I[t] = ∅
     if t is monosemous then
          I[t] := the only sense of t
     else
          P := P U {t}
     end if
end for



for each {i ∈ I} do
      for each {p ∈ P} do
            for each {sense j ∈ p} do
                  d[i, j] = dijkstra_shortest_path(i, j)
            end for
     end for
end for

repeat
      P' := P
      for each {p ∈ P} do
            BestSense := ∅
            MaxValue := 0
            for each {sense j ∈ p} do
                  W[s] := 0
                  N[s] := 0
                  for each {sense i ∈ I} do
                        w := d[i,j]
                        if w > 0 then
                              W[s] := W[s] + (1/w)
                              N[s] := N[s] + 1
                        end if
                end for
                if N[s] > 0 then
```

```
                    NewValue := W[s]/N[s]
                    if NewValue > MaxValue then
                            MaxValue := NewValue
                       BestSense := s
                       end if
                end if
            end for
            if MaxValue > 0 then
                    I[t] := BestSense
                    P := P \ {t}
                    i := BestSense
                    for each {p ∈ P} do
                          for each {sense j ∈ p} do
                                d[i, j] = dijkstra_shortest_path(i, j)
                          end for
                    end for
            end if
      end for
until P ≠ P'
return (I,P);
```

Algoritmo 2: SSI-Dijkstra-Fast

5.4.- Diagrama UML

Ya que se han generado clases nuevas, que hay que integrar en UKB, un diagrama UML puede ilustrar de forma simple el estado actual de la aplicación. Hay que tener en cuenta que no se va a representar absolutamente todo el diagrama de clases, sino únicamente la parte de aquellas clases que estén relacionadas de forma directa con las nuevas clases creadas.

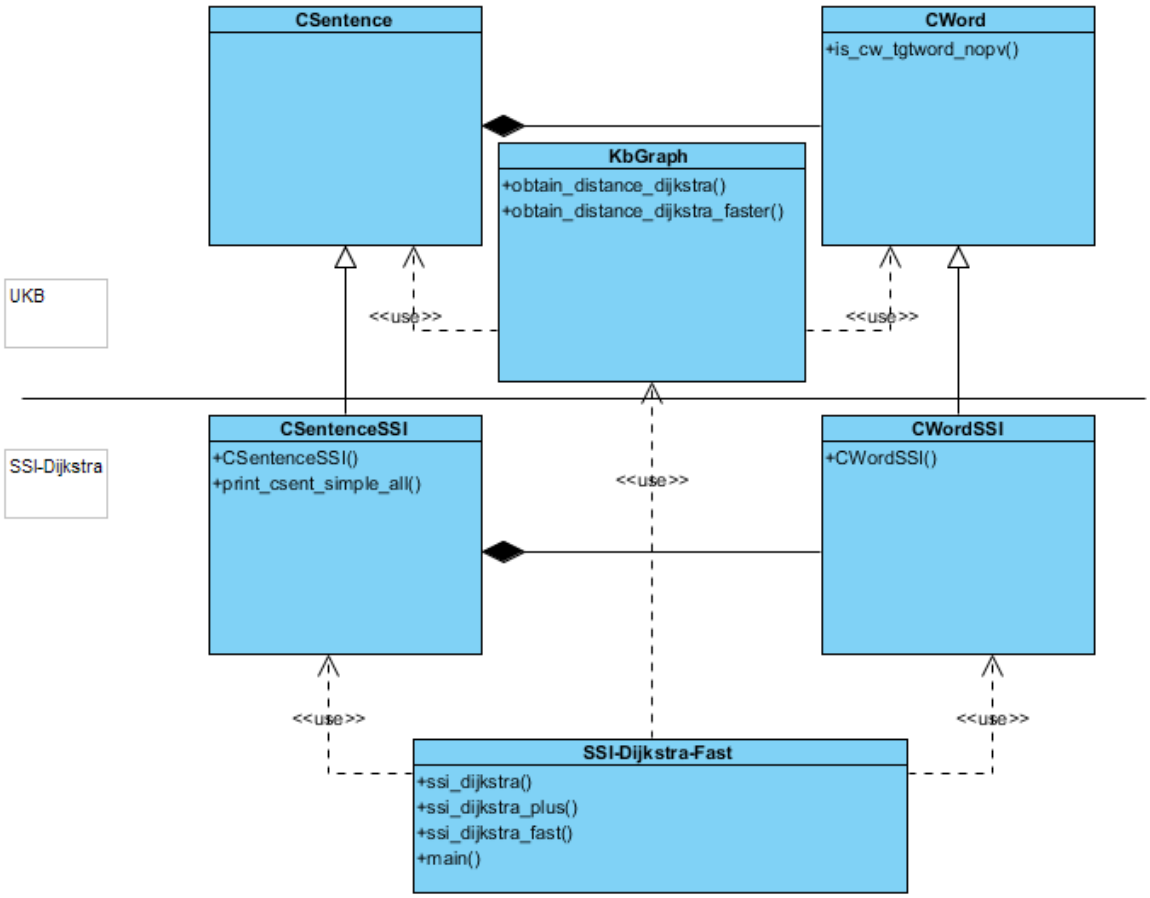


Imagen 4: Diagrama UML (imcompleto)

6.- Diseño

6.1.- Pseudocódigos

Se pretende dar una idea general de cómo se han codificado los algoritmos utilizando pseudocódigo. Los algoritmos que no hayan sufrido cambios importantes en este proyecto, no se describirán.

6.1.1.- SSI-Dijkstra-Fast

Descripción: Es una función que recibiendo un contexto de entrada genera una nueva estructura de datos de contexto con las palabras del contexto de entrada desambiguadas.

Pseudocódigo:

```
SSI_Dijkstra_Fast (contexto)
// Inicialización: parte 1
para cada palabra ∈ contexto
      caso(palabra.significados())
            caso 0: // Algo extraño ha pasado
                        romper;
            caso 1: // Palabra monosémica
                        I.añadir(palabra.significados()[0])
                        solucion.añadir(palabra)
                        romper;
            defecto: // Palabra polisémica
                        P.añadir(palabra)
                        romper;
      fin-caso
fin-repetir

// Inicialización: parte 2 (FSI)
si I ≠ ∅ entonces
      palabraPtoI = P[0].significados()
      i = 1
      mientras i ≠ P.tamaño()
            total = 0
```

```
            synsets = 0
            palabraP = P[i]
            significadoPtoI =
                        palabraPtoI.significados()[0]
            para cada significadoP ∈ palabraP
                  w =   obtener_distancia(significadoP,
                        significadoPtoI, grafo)
                  si w > 0
                        total = total + (1/w)
                        synsets++
                  fin-si
            fin-para
            si synsets > 0
                  v = total/synsets
                  si v > max
                        max = v
                        best = significadoPtoI
                  fin-si
            fin-si
      fin-mientras
      si max > 0
            I.añadir(best)
            solucion.añadir(new Palabra(best))
            P.borrar(P[0])
      fin-si
fin-si
matriz = crear_matriz(I, P)
// Paso iterativo
si I ≠ ∅ entonces
      mientras P ≠ ∅
            vector_P = P[0].significados()
            total = 0
            synsets = 0
            para cada significadoP ∈ P
                  para cada significado'I ∈ I
                        w = matriz[significado'I , significadoP]
                        si w > 0
                              total = total + (1/w)
                              synsets++
```

```
                        fin-si
                  fin-para
                  si P[0].pos() = 'v' // FSP
                        i = 1
                        mientras i ≠ P.tamaño()
                              palabra = P[i]
                              significadoP =palabra.significados()[0]
                              w =
                                    obtener_distancia(significadoP,
                                          significado'I)
                              si w > 0
                                    total = total + (1/w)
                                    synsets++
                              fin-si
                        fin-mientras
                  fin-si // Fin FSP
                  si synsets > 0
                        v = total/synsets
                        si v > max
                              max = v
                              best = significado'I
                        fin-si
                  fin-si
            fin-para
            si max > 0
                  I.añadir(best)
                  solucion.añadir(new Palabra(best))
                  P.borrar(P[0])
                  matriz = actualizar_matriz(best, P)
            fin-si
      fin-mientras
fin-si
devolver solucion
```

7.- Implementación

7.1.- Sobre UKB: lo que no teníamos

Como ya comentamos en el apartado de Análisis, había cosas de UKB que necesitábamos crear ya que el propio software no nos las proporcionaba. En este apartado, se va a intentar explicar de una forma un poco más concreta qué se ha añadido a UKB, cómo se ha hecho y por qué se ha tenido que hacer. Para poder explicar qué se ha añadido, es necesario explicar un poco por encima cuáles son las estructuras de datos y métodos básicos de UKB relacionados con este proyecto.

7.1.1.- Contextos como estructuras de datos: CSentence

El CSentence es la estructura propia de UKB para guardar los datos asociados a un contexto. En sí, un CSentence no contiene más que un string (en el que guardamos el id del contexto) y un vector de CWord. De forma gráfica, podría verse como algo así:

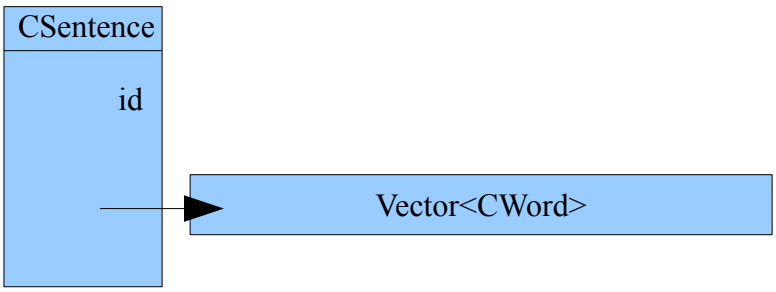


Imagen 5: Estructura de un CSentence

Un CWord, a su vez, es una estructura de datos en la que guardamos toda la información asociada a una palabra. Esta información va algo más allá de lo que necesita el SSI-Dijkstra en realidad (habrá información de un CWord que no se usará en nuestro algoritmo), pero sí que posee toda la información que necesitamos, por lo que es idónea para este trabajo. Además, se le suma la ventaja de que tenemos funciones tanto de lectura como de escritura sobre algunos de sus parámetros, todo ello por supuesto orientado a UKB.

Un CWord contiene (además de otras cosas) un string (donde guardaremos la palabra a desambiguar), un carácter (donde guardaremos el PoS de la palabra) y una lista de strings asociados a los sentidos propios palabra (o lo que es lo mismo, una lista que representa algunos nodos en el

grafo). Como se trata de ver cuál es el mejor sentido de la palabra (o lo que es igual, de decidir qué nodo de la lista de nodos se amolda mejor al contexto que estamos tratando), en principio con esto debería valernos.

CWord cuenta además con un atributo booleano (que indica si la palabra está desambiguada o no), con un peso propio de la palabra (que venía ya con UKB y que permite la ordenación explícita) y también con un identificador propio de la palabra (lo cual permite identificar la palabra dentro del contexto con un identificador cualquiera).

En la versión 1.6 de UKB (y por tanto, en 2.0 también) se introdujo un atributo que indicaba el tipo de la palabra, a diferenciar entre:

- *cw_ctxword:* palabras que se desambiguan, ayuda a desambiguar otras, pero no se muestran en el resultado.
- *cw_tgtword:* palabras que se desambiguan (son mayoría en contextos normales).
- *cw_concept:* palabras que no se desambiguan, pero afectan en el resultado de otras (synsets)
- *cw_tgtword_nopv:* palabras que se desambiguan, pero no afectan en el resultado de otras.

Una vez planteadas las bases sobre lo que se va a trabajar, es posible empezar a describir sus características y sus limitaciones a la hora de desarrollar este proyecto concreto.

7.1.2.- Limitaciones: creación de nuevos métodos y clases

Se estudiaron tanto la estructura de datos como los métodos del propio UKB. Respecto a las estructuras de datos utilizadas, eran suficientes para el desarrollo de este proyecto.

Pero SSI-Dijkstra requiere trabajar con todos los tipos de palabra (incluido *cw_tgtword_nopv,* palabras que el constructor de UKB convierte automáticamente en palabras de tipo *cw_tgtword* sin peso, ya que UKB no trabaja directamente con ellas), y queríamos aprovechar también la función de salida (mostrado de datos) que nos permitía elegir de qué forma mostrar algunos datos (esto son en realidad problemas viejos [Ledesma, 2012] a los que se les ha dado otra solución diferente para

impedir colisionar con los métodos de UKB).

Para solucionar este problema, se han creado dos clases nuevas CSentenceSSI y CWordSSI, las cuales heredan directamente todos sus atributos y métodos de CSentence y CWord. Para conseguir que las clases trabajen directamente con los atributos de UKB (evitando así programar varios *getters* y *setters* directamente en UKB), se ha cambiado la visibilidad de algunos atributos de ambas clases de UKB a *protected.* Para conseguir los nuevos métodos de creación y salida de datos, se han programado directamente en las nuevas clases.

Con estas dos nuevas clases hemos podido definir las siguientes cosas:

a) Creación de un nuevo método para mostrar los resultados (*print_csent_simple_all*). Para poder adaptar el algoritmo a las diferentes formas de mostrar resultados, vimos necesario crear una función para poder mostrarlos de forma independiente de la de UKB en CsentenceSSI.

b) Un constructor específico para CWordSSI. El que tiene UKB no permite crear directamente palabras desambiguadas, sino simplemente decidía si estaban desambiguadas o no según el número de sentidos. El nuevo constructor permite crear una nueva palabra ya desambiguada. La razón de ello es que el algoritmo SSI-Dijkstra trabaja con vectores de CWordSSIs, tanto en las palabras por desambiguar como en la solución general del problema.

c) Un constructor específico para CSentenceSSI. Dado que el programa trabaja directamente con estructuras CSentenceSSI (lectura y escritura de UKB), era necesario generar una estructura de datos de este tipo con todas las palabras desambiguadas. Por eso, se creó un nuevo constructor que admite un id y un vector de CWordSSI como parámetros de entrada (ya que el método de SSI-Dijkstra nos genera un vector de CWordSSI con todas sus palabras desambiguadas y el id podemos cogerlo del CSentenceSSI de entrada).

De todas formas, y a pesar de la creación de estas dos nuevas clases se ha visto necesario incluir algunos métodos extra en UKB:

a) Un método nuevo *is_cw_ctxword* que indica si la palabra a tratar es de tipo *cw_ctxword.* Esto era necesario para realizar una correcta distinción en los constructores nuevos.

b) Un método nuevo *is_cw_tgtword_nopv* que indica si la palabra a tratar es de tipo *cw_tgtword_nopv.* Esto era necesario para el correcto funcionamiento de la función que muestra resultados definida en CSentenceSSI.

7.2.- Nuevas soluciones a problemas antiguos

7.2.1.- Cambios en la estructura nuclear de UKB

El problema fundamental que surgió cuando quisimos empezar a trabajar con la nueva versión de UKB, es que debido al cambio en la estructura de datos nuclear de la herramienta ahora había operaciones que no podíamos realizar de forma directa.

La nueva clase que implementa la estructura de datos del grafo impide realizar operaciones de modificación (en principio, aunque al parecer hay otras operaciones no permitidas). También hay problemas con la función BoostGraph que utilizamos para calcular el algoritmo de Dijkstra desde un origen.

Para solucionar este problema ha sido necesario incluir las funciones de cálculo de distancias entre dos nodos del grafo en la clase *KbGraph* de UKB (en lugar de en la propia clase principal del proyecto, como habíamos hecho hasta ahora), y gracias a un pequeño truco (cortesía de Aitor Soroa) hemos podido hacer lo necesario para trabajar correctamente con estas funciones.

Debido a ello, estas funciones (y los algoritmos anteriormente implementados SSI-Dijkstra y SSI-Dijkstra+) han sufrido pequeños cambios y no necesitan el grafo como parámetro (lo usan directamente cuando hacen la llamada a estas funciones). A pesar de ello, la eficiencia y la corrección de los resultados no se ha visto comprometida.

7.2.2.- Opción *poly*. Ordenación por polisemia

Versiones anteriores de SSI-Dijkstra implementan una opción de ordenación por polisemia (*poly*). Al introducir el nuevo atributo que indica el tipo de palabra que estamos procesando, el *sort* de C++ daba un problema: el algoritmo ordenaba correctamente las palabras por polisemia, pero no ordenaba los tipos; de forma que palabras que inicialmente tenían un tipo acababan con

(probablemente) otro tipo diferente: el correspondiente al de la palabra anteriormente situada en esa posición en el vector de CWord del CSentence.

Para solventar este problema, se ha visto necesario realizar una modificación en el método operator= (asignación) de la clase CWord. Se ha visto necesario añadir una condición para que los tipos de cada una de las palabras también sean asignados y así poder utilizar correctamente el método de ordenación.

7.3.- Especificaciones de proyectos anteriores

Tal y como se indica en el apartado de Captura de Requisitos, este nuevo proyecto permite al usuario trabajar con las herramientas con las que podía trabajar en versiones anteriores. Esto es, el usuario es capaz de decidir (a base de opciones en el programa):

a) El algoritmo con el que quiere trabajar.

b) Si quiere utilizar la optimización o no en sus operaciones.

c) El tipo de algoritmo de ordenación con el que quiere operar.

8.- Pruebas

Uno de los objetivos del proyecto, es probar que lo realizado funciona correctamente (al menos, tan correctamente como las versiones anteriores de SSI-Dijkstra) y que además, existe una mejora en la velocidad de los resultados . Para ello se han realizado dos tipos de pruebas: pruebas de validez de resultados y pruebas de rendimiento temporal.

Para las pruebas se han utilizado dos conjuntos de testeo (contextos), uno de ellos conformado por contextos con muchas palabras desambiguadas y unas pocas palabras a desambiguar[20], y otro de ellos conformado por contextos con muchas palabras por desambiguar[21] (mucho más cercano a lo que podría ser un contexto en un entorno normal de ejecución).

8.1.- Validez de resultados

Para comprobar la validez de los resultados, y para ilustrar la similaridad de los mismos en la nueva herramienta, se ha decidido realizar una prueba de validez de resultados. Consiste básicamente en: utilizando como contextos de pruebas el que tiene pocas palabras pendientes de desambiguar y como herramienta de testeo el scorer2 del Senseval; comprobar la precisión de ambos algoritmos y tratar de ver que realmente son similares.

Antes de pasar a analizar las pruebas, debemos definir tres medidas de calidad de la solución presentes en el análisis:

Precision (P): Se define como el número de contextos correctos del total de contextos con solución.
Recall (R): Se define como el número de contextos correctos del total de contextos del fichero. Es lógico pensar que este número será siempre menor o igual que P.
Attempted (A): Se define como el porcentaje de contextos que la máquina ha intentado desambiguar (podría ser que existan contextos que no puedan ser desambiguados usando un algoritmo concreto).

20 http://adimen.si.ehu.es/~rigau/wnet30+g.v3.random.contexts
21 http://adimen.si.ehu.es:/~rigau/wnet30+g.v3.random.1.contexts

8.1.1.- SSI-Dijkstra+ 1.6 vs SSI-Dijkstra+ 2.0

	Precision	Recall	Attempted
SSID 1.6	0,905	0,826	91,32 %
SSID 2.0	0,905	0,826	91,32 %

8.1.2.- SSI-Dijkstra-Fast 1.6 vs SSI-Dijkstra-Fast 2.0

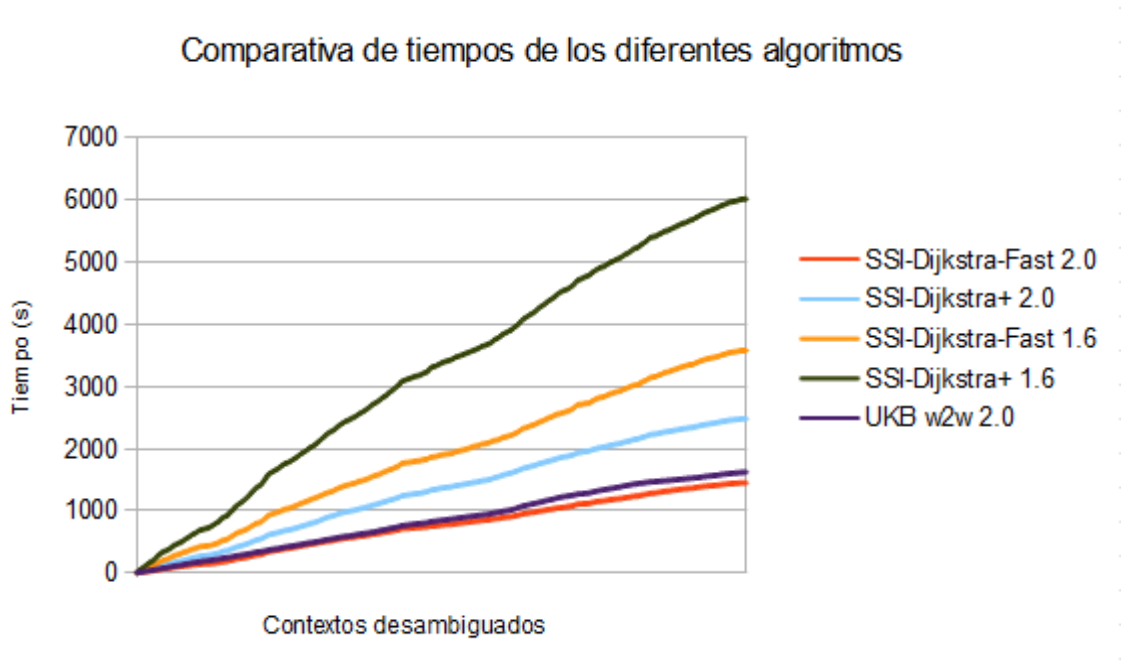
	Precision	Recall	Attempted
SSID 1.6	0,905	0,826	91,32 %
SSID 2.0	0,905	0,826	91,32 %

8.1.2.- UKB w2w 2.0

	Precision	Recall	Attempted
UKB 2.0	0,904	0,82	91,10%

8.2.- Velocidad de resultados

Para comprobar la velocidad de los resultados, se ha realizado una pasada de la ejecución de los diferentes algoritmos y tomando muestras de cada ejecución del tiempo de los mismos se ha realizado el siguiente gráfico.



Como se puede comprobar, los algoritmos de versión 2.0 son más rápidos que los de versión 1.6, incluso el algoritmo SSI-Dijkstra+ con la versión 2.0 de UKB es más rápido que el algoritmo SSI-Dijkstra-Fast con la versión 1.6 de UKB. De aquí se puede deducir, que, aunque los algoritmos Fast son más eficientes temporalmente que los algoritmos Plus, el cambio de versión de UKB de 1.6 a 2.0 ha supuesto también una importante mejora en el rendimiento temporal de los algoritmos (incluso más que la presente entre las versiones Plus y Fast de SSI-Dijkstra).

También se ha hecho una pequeña prueba con la opción más precisa (pero más lenta) de UKB 2.0, y se puede ver que es aproximadamente igual de rápido que la versión más rápida de SSI-Dijkstra.

9.- Gestión

9.1.- Esfuerzo total planificado vs real

Como en todo proyecto, la planificación inicial no tiene por qué ajustarse a la realidad de los plazos y del tiempo que se invierte en cada cosa. En los siguientes apartados, así como las siguientes tablas y gráficos ilustran la diferencia entre el esfuerzo planificado y el real:

9.1.1.- En procesos tácticos

Proceso	Tiempo planificado	Tiempo real	Desviación relativa(%)
Planificación	25 horas	20 horas	-20,00%
Gestión	15 horas	15 horas	0,00%
Instalación herramientas	10 horas	10 horas	0,00%
Reuniones	30 horas	25 horas	-16,67%
Integración	10 horas	10 horas	0,00%
TOTAL:	**90 horas**	**80 horas**	**-11,11%**

Tabla 2: Comparativa planificado vs real: procesos tácticos

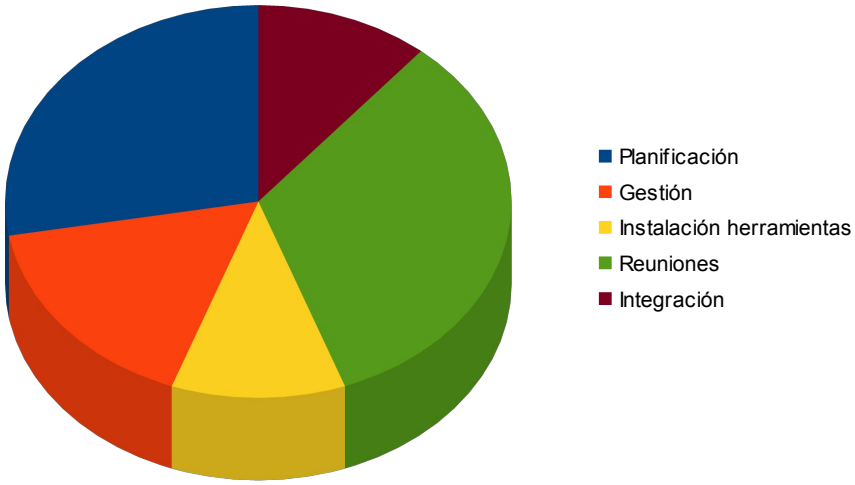


Imagen 6: Esfuerzo planificado en procesos tácticos

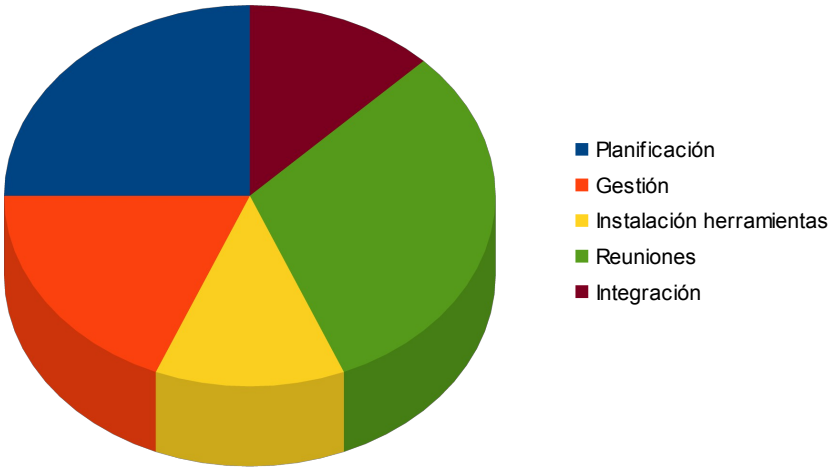


Imagen 7: Esfuerzo real en procesos tácticos

En la tabla 3 y en la imágenes 5 y 6, mostramos la comparativa entre el esfuerzo planificado y el esfuerzo real en los procesos tácticos. Se puede comprobar que en este proyecto se ha planificado más cantidad en estos procesos de la realmente necesaria (aunque la diferencia en realidad no es especialmente alta).

9.1.2.- En procesos operativos

9.1.2.1.- Primera iteración

Proceso	Tiempo planificado	Tiempo real	Desviación relativa (%)
Captura requisitos	5 horas	5 horas	0,00%
Análisis	5 horas	5 horas	0,00%
Diseño	5 horas	8 horas	60,00%
Implementación	10 horas	15 horas	50,00%
Pruebas	8 horas	10 horas	25,00%
TOTALES:	**33 horas**	**43 horas**	**30,30%**

Tabla 3: Comparativa planificado vs real: procesos operativos - 1era iteración

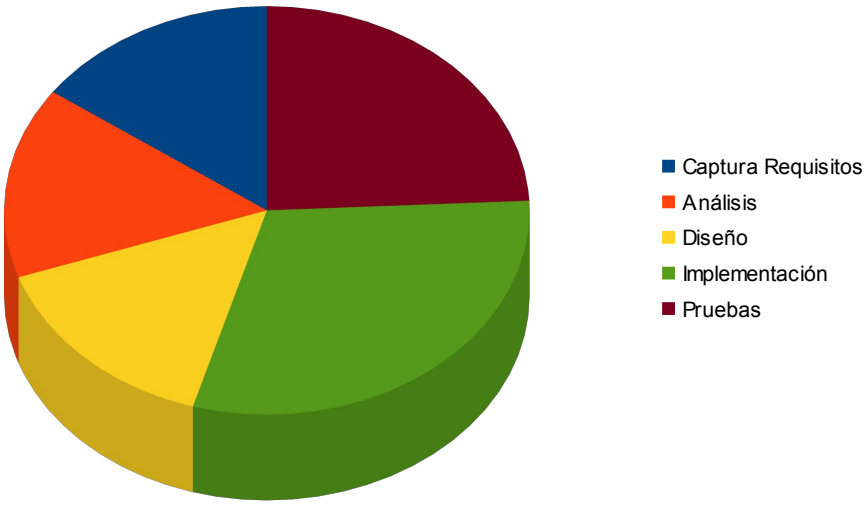


Imagen 8: Esfuerzo planificado en procesos operativos – 1ª iteración

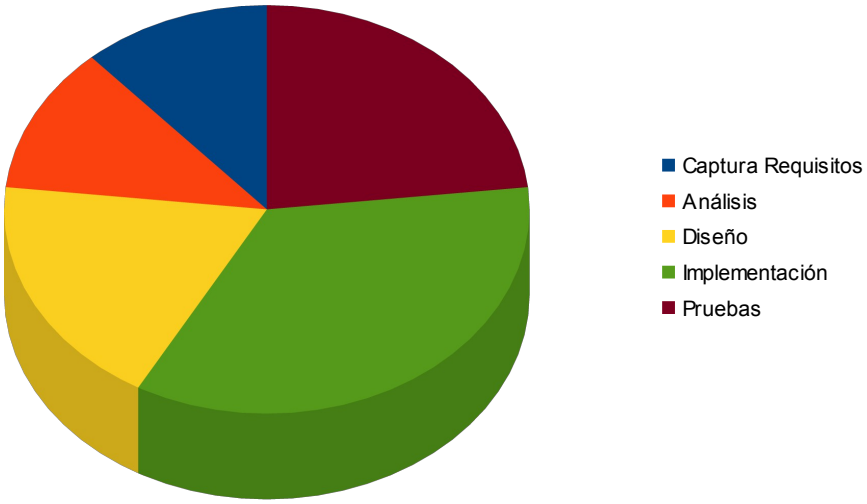


Imagen 9: Esfuerzo real en procesos operativos – 1ª iteración

En la tabla 4 y en la imágenes 7 y 8, mostramos la comparativa entre el esfuerzo planificado y el esfuerzo real para la primera iteración de los procesos operativos. Se puede observar que las partes de diseño, implementación y pruebas de esta iteración han llevado más tiempo del inicialmente planificado, básicamente por los problemas anteriormente comentados en el apartado de implementación.

9.1.2.2.- Segunda iteración

Proceso	Tiempo planificado	Tiempo real	Desviación relativa (%)
Captura requisitos	2 horas	2 horas	0,00%
Análisis	2 horas	3 horas	50,00%
Diseño	5 horas	3 horas	-40,00%
Implementación	15 horas	20 horas	33,33%
Pruebas	2 horas	10 horas	400,00%
TOTALES:	**26 horas**	**38 horas**	**46,15%**

Tabla 4: Comparativa planificado vs real: procesos operativos - 2da iteración

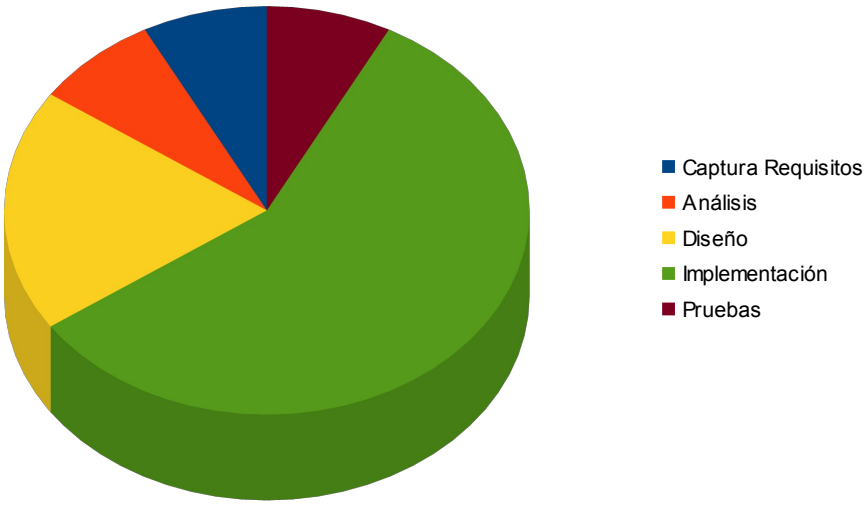


Imagen 10: Esfuerzo planificado en procesos operativos – 2ª iteración

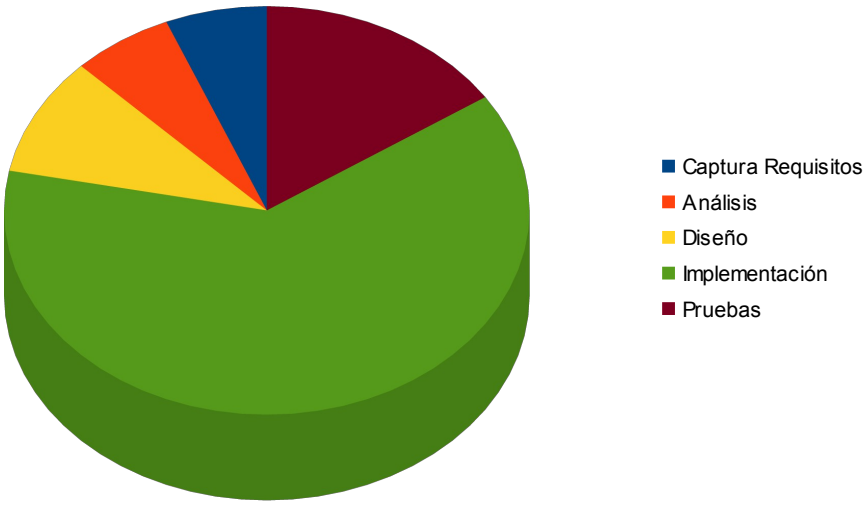


Imagen 11: Esfuerzo real en procesos operativos – 2ª iteración

En la tabla 5 y en la imágenes 9 y 10, mostramos la comparativa entre el esfuerzo planificado y el esfuerzo real para la segunda iteración de los procesos operativos. Se puede ver una diferencia importante especialmente en la parte de pruebas, que llevó más tiempo del que inicialmente se había pensado.

9.1.3.- En procesos formativos

Proceso	Tiempo planificado	Tiempo real	Desviación relativa (%)
Lenguaje C++	5 horas	5 horas	0,00%
BoostGraph	5 horas	5 horas	0,00%
UKB 2.0	15 horas	20 horas	33,33%
Proyectos anteriores	20 horas	10 horas	-50,00%
Memoria	60 horas	70 horas	16,67%
Presentación	10 horas	Desconocido	Desconocido
Integración	10 horas	Desconocido	Desconocido
TOTALES:	**125 horas**	**90 horas**	**125,00%**

Tabla 5: Comparativa planificado vs real: procesos formativos

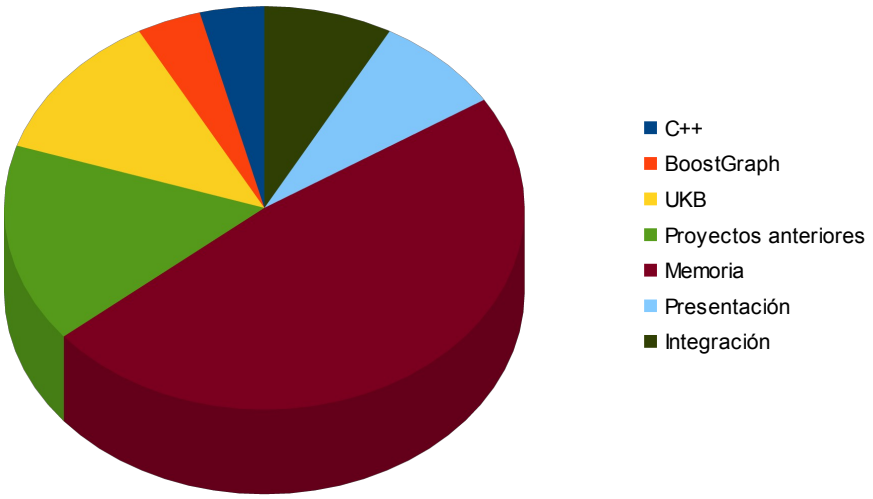


Imagen 12: Esfuerzo planificado en procesos formativos

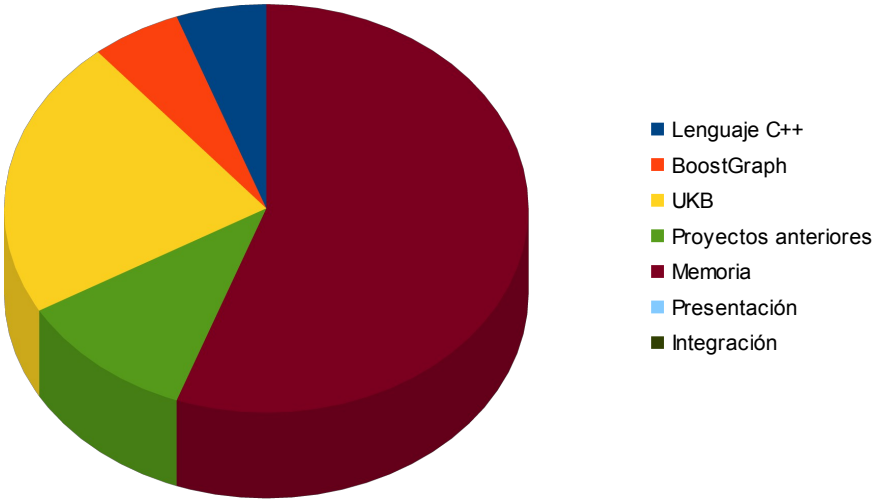


Imagen 13: Esfuerzo real en procesos formativos

En la tabla 7 y en la imágenes 13 y 14, mostramos la comparativa entre el esfuerzo planificado y el esfuerzo real para los procesos formativos.

9.1.4.- Un punto de vista general

En general, se puede observar que la mayor parte de las desviaciones importantes se dan en procesos operativos, en su mayoría por problemas con los que inicialmente no se contaban y a

consecuencia de ello, aparece una tendencia a que estos procesos lleven más tiempo del planificado.

9.2.- Justificación de las desviaciones

La mayoría de las desviaciones son consecuencia de la aparición de problemas inesperados en el desarrollo software del proyecto[22] (se puede ver que la mayoría de las desviaciones importantes se dan en esos procesos). Debido a esto, muchos de estos procesos han tenido que ser alargados más allá de lo que realmente se esperaba en la planificación inicial.

9.3.- Incidencias principales

No han existido incidencias externas durante el desarrollo del proyecto.

No hay incidencias excepto las ya comentadas en la sección 9.2.

22 Ya comentadas en la sección de Implementación.

10.- Conclusiones

10.1.- Objetivos cumplidos

Considerando los objetivos propuestos en el Documento de Objetivos del Proyecto:
1.- Recodificar los algoritmos SSI-Dijkstra y SSI-Dijkstra+ en C++ (haciéndolos compatibles con la versión 2.0 de UKB).
2.- Codificar el algoritmo SSI-Dijkstra-Fast utilizando las librerías BoostGraph y el software UKB para ello.
3.- Conseguir que este proyecto cumpla también las especificaciones relacionadas con proyectos anteriores.

Creo que se han cumplido las metas marcadas. Los objetivos marcados inicialmente están cubiertos: no solo tenemos una aplicación escrita en C++ que realiza el mismo trabajo, sino que ademas se acopla perfectamente con el software de UKB.

10.2.- Valoración personal

La primera impresión que tuve con este proyecto fue que no iba a ser una cosa especialmente larga, ya que habiendo trabajado con algoritmos relacionados y cercanos a éste, pensé que sería más fácil de lo que al final ha resultado ser.

Pero igual, estoy satisfecho con los resultados ya que por segunda vez he conseguido hacer algo útil y funcional, y esta vez, está integrado como parte de la herramienta UKB, lo cual permite que este algoritmo tenga bastante más visibilidad y más gente pueda utilizarlo.

Además, me agrada el campo de trabajo de este proyecto y me gustaría seguir trabajando en cosas relacionadas con él en un futuro.

10.3.- Trabajos futuros

En este apartado presentaremos algunos ejemplos de cosas que podrían ser trabajos futuros relacionados con este proyecto y mejoras en alguno de sus aspectos.

10.3.1.- Implementar otros algoritmos que usan Dijkstra

Podría ser interesante que, una vez tenemos la función que calcula las distancias mínimas en C++, podamos usarla para implementar otros algoritmos que utilicen el algoritmo de Dijkstra como base. La implementación actual con el uso de distancias mínimas, permite pensar en explorar nuevos tipos de algoritmos de desambiguación.

10.3.2.- Usar otras funcionalidades de BoostGraph que no sean Dijkstra

Usar otros métodos ya implementados en BoostGraph para calcular distancias mínimas. Por ejemplo, usar el Bellman's-Ford Shortest Paths en vez de Dijkstra.

10.3.3.- Comparativa con una solución escrita en otro lenguaje

Sería interesante comprobar el rendimiento temporal de la aplicación si en lugar de C++ utilizara otro lenguaje de programación (por ejemplo, Perl o Python).

10.3.4.- UKB como servicio cliente-servidor

Ya que la lectura del grafo de UKB y la lectura del diccionario tarda un tiempo importante (normalmente determinante para ficheros de contextos pequeños), sería interesante tener un *daemon* que tenga cargados estos ficheros y que atienda las peticiones de los clientes para intentar mejorar la eficiencia global del programa.

11.- Bibliografía

11.1.- Artículos

- [Navigli & Velardi, 2005] Structural Semantic Interconnections: A Knowledge-Based Approach to Word Sense Disambiguation. disambiguation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 27:1075-1086.
- [Cuadros & Rigau, 2008] KnowNet: Building a Large Net of Knowledge from the Web. 22nd International Conference on Computational Linguistics COLING'08. Manchester, UK. 2008.
- [Laparra et al., 2010] Exploring the Integration of WordNet and FrameNet. Proceedings of the 5th Global WordNet Conference (GWC'10), Mumbai, India. January, 2010.
- [Cuadros & Rigau, 2007] Bases de Conocimiento Multilíngües para el Procesamiento Semántico a Gran Escala. Acceso y visibilidad de la información multilingüe en la red: el rol de la semántica, pg. 25--52. 2008, UNED, Ed Felisa Verdejo Maillo y Ana García Serrano, ISBN 978-84-362-5609-3
- [Miller et al., 1990] Miller, G., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. (1990). Five Papers on WordNet. Special Issue of International Journal of Lexicography, 3(4).
- [Fellbaum, 1998] WordNet: An Electronic Lexical Database. The MIT Press.
- [Vossen, 1998] EuroWordNet: A Multilingual Database with Lexical Semantic Networks. Kluwer Academic Publishers.
- [Agirre & Soroa, 2009] Personalizing PageRank for Word Sense Disambiguation. Proceedings of the 12th conference of the European chapter of the Association for Computational Linguistics (EACL-2009). Athens, Greece
- [Atserias et al., 2004] The MEANING multilingual central repository. In Proceedings of the Second International Global WordNet Conference (GWC'04) Brno, Czech Republic, January 2004, ISBN 80–210–3302–9.
- [Rigau el al., 2002] Rigau G., Magnini B., Agirre E., Vossen P. and Carroll J., MEANING: A Roadmap to Knowledge Technologies. Proceedings of COLING Workshop "A Roadmap for Computational Linguistics". Taipei, Taiwan. 2002.

11.2.- Libros

- [Agirre & Edmonds, 2006] Word Sense Disambiguation: Algorithms and Applications
- [Cormen et al., 2001] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGraw-Hill. pp. 595–601. ISBN 0-262-03293-7.

11.3.- Otros proyectos

- [Amenabar, 2010] Desarrollo de aplicación web para la desambiguación de sentidos de palabras para la evocación (Xabier Aramendi Amenabar, San Sebastián, 2010)
- [Blanco, 2010] SSI-Dijkstra+: Desarrollo de un sistema de resolución de la ambigüedad semántica basada en grafos (Gorka Blanco Gutierrez, San Sebastián, 2010)
- [Ledesma, 2012] SSI-Dijkstra+: Desarrollo de un sistema de resolución de la ambigüedad semántica basada en grafos (Aritza Ledesma Ruiz, San Sebastián, 2012)
- [Chihuailaf, 2013] Optimización del algoritmo de WSD SSI-Dijkstra (Rubén Chihuailaf Muñoz, Chile, 2013)