ON THE IMPLEMENTATION OF AN INference
ALGORITHM IN JAVA

Approved by:

Dr. Dan Moldovan

Dr. Sanda Harabagiu

Dr. Margaret Dunham

Dr. Jacob Kornerup

Dr. Peter Raad
ON THE IMPLEMENTATION OF AN INFERENCE ALGORITHM IN JAVA

A Dissertation Presented to the Graduate Faculty of the School of Engineering and Applied Science Southern Methodist University in Partial Fulfillment of the Requirements for the degree of Masters of Science with a Major in Computer Science by

Vasile Rus (B.Sc., Technical University of Cluj-Napoca, 1997)

May 15, 1999
ACKNOWLEDGMENTS

My deepest appreciation goes to my advisor, Professor Dan I. Moldovan, for his help in my graduate studies, guidance and encouragement when I was in trouble, which made it possible to achieve this work.

I would also like to thank the members of my committee Dr. Sanda M. Harabagiu for her advice and technical help, Dr. Margaret Dunham, Dr. Jacob Kornerup, Dr. Frank Coyle and Dr. Peter Raad.

I am very grateful to the faculty, staff and students of the Department of Computer Science for their help and support.
On the Implementation of an Inference Algorithm in Java

Advisor: Professor Dan Moldovan

Masters of Science degree conferred May 15, 1999
Dissertation completed May 15, 1999

This masters thesis presents an implementation of an inference algorithm based on marker propagation processing model. Marker propagation is a technique that was developed for using parallelism to find connections between concepts in semantic networks.

All the previous implementations of the marker propagation are either in hardware or in low level programming languages. This lacks flexibility and implies a considerable effort to develop a marker passer for a new problem. In this thesis we take advantage of the abstraction and encapsulation properties of the object-oriented technology to implement a framework for marker propagation that deals with the general issues of the marker model, leaving the specific aspects of an individual problem to be implemented by the user of the framework, by specializing it.

A contribution of this thesis is the usage of threads for markers. We regard the semantic network as a network of hosts and the markers as threads that run in the context of a host. The markers propagate from one host to another according to their propagation rules. Once in a host, they are scheduled to execution using a local customable scheduling algorithm. The default one is a first-in-first-out scheduler.

Two basic applications were developed: text inference and graph matching.

The pattern matching application consists of finding matchings of a small pattern graph in a larger one. The network is loaded in memory ahead of time and then the propagation of markers starts. For the case when the semantic network fits in the machine’s memory we may find out the matchings in a time linear with the length
of the critical path $P$, the longest sequential path in the pattern. When the semantic network is too big to fit into memory we could save the already processed nodes on the disk but this would require extra disk accesses. Instead, we propose a heuristic that increases the size of the in-memory base with 33%, avoiding any extra accesses to disk.

Text inference means extracting information unstated in a text, but implied. The inferences are based on world knowledge. We use the algorithm developed in [19], which uses WordNet, an electronic lexical dictionary, as world knowledge. The algorithm consists of highly parallel search methods that would be hardly handled using conventional processing models. It tries to find semantic paths between concepts in a given text. Most of the paths found are irrelevant. To filter them we use methods presented in [19] that keep only those paths that explain the lexical relations between concepts. We prove the effectiveness of those methods and show the inferences obtained. There are two main implementation issues for this application: the marker spawning and knowledge base activation. The marker spawning is significant and the markers generated dominate the memory resources. On the other side, the knowledge base is big and it would be impossible to load it into memory ahead of time. To solve these two problems a heuristic to reduce the number of markers spawned is proposed, respectively, a lazy loading procedure of the nodes of the knowledge base is applied. The speedup of the marker propagation approach over the serial case would be almost 50 had we enough processors to run all threads in parallel.

Java is used as a development environment because it has object-oriented features and multithreading capabilities that support the parallelism offered by the marker propagation model. A Java server was developed in order to enable Java applications retrieve all data available in WordNet.

The memory requirements are the main drawback of our system which we believe will be solutioned in a distributed space at the expense of time complexity. Another possible weakness is due to the performance of Java. It may be improved using a just-in-time compiler.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ......................................................... iii

LIST OF FIGURES ............................................................. viii

LIST OF TABLES ............................................................... x

CHAPTER

1. INTRODUCTION ............................................................. 1
   1.1. Motivation ............................................................ 1
   1.2. General Goals ....................................................... 2
   1.3. Thesis Outline ...................................................... 2

2. OBJECT-ORIENTED MARKER PROPAGATION ......................... 4
   2.1. Object-Oriented Systems .......................................... 4
       2.1.1. Definitions ................................................. 4
   2.2. Marker Propagation Model ........................................ 7
   2.3. Threaded Object-based marker propagation .................... 11
       2.3.1. Semantic Network Abstraction ............................. 13
       2.3.2. The Markers ................................................. 15
   2.4. Marker propagation in Java ....................................... 19
       2.4.1. Java Development Environment ............................. 20
           2.4.1.1. Java - the language ................................. 20
           2.4.1.2. Java Performance ..................................... 20
       2.4.2. Java Threads ................................................ 22
           2.4.2.1. Synchronization ....................................... 22
           2.4.2.2. The Thread class ..................................... 24
2.4.2.3. Thread controls.............................. 24

2.4.3. Marker Propagation Implementation in Java ............... 25

2.5. Summary ............................................. 27

3. PATTERN MATCHING APPLICATION............................ 28

3.1. Application description.................................... 28

3.2. Implementation and Results.................................. 32

4. TEXT INference ........................................ 39

4.1. The problem ........................................... 39

4.1.1. The Knowledge Base.................................. 40

4.1.2. Algorithm Assumptions and Approach...................... 41

4.1.3. Parallel Search Engine .................................. 42

4.2. An example ............................................ 45

4.3. Design Issues ........................................... 46

4.4. Experiments ............................................. 52

5. CONCLUSIONS ........................................... 54

REFERENCES ............................................. 57
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.</td>
<td>Object Encapsulation (from [42])</td>
<td>5</td>
</tr>
<tr>
<td>2.2.</td>
<td>Semantic Network - adapted from [40]</td>
<td>8</td>
</tr>
<tr>
<td>2.3.</td>
<td>SNAP - Marker Propagation Implementation (from [40])</td>
<td>12</td>
</tr>
<tr>
<td>2.4.</td>
<td>Object-based Marker Propagation Model</td>
<td>13</td>
</tr>
<tr>
<td>2.5.</td>
<td>Semantic Network</td>
<td>14</td>
</tr>
<tr>
<td>2.6.</td>
<td>Host generic object</td>
<td>14</td>
</tr>
<tr>
<td>2.7.</td>
<td>Host generic object</td>
<td>15</td>
</tr>
<tr>
<td>2.8.</td>
<td>Marker spreading</td>
<td>17</td>
</tr>
<tr>
<td>2.9.</td>
<td>Marker object</td>
<td>19</td>
</tr>
<tr>
<td>2.10.</td>
<td>Java computing model</td>
<td>21</td>
</tr>
<tr>
<td>2.11.</td>
<td>Java Thread class</td>
<td>25</td>
</tr>
<tr>
<td>3.1.</td>
<td>Pattern</td>
<td>29</td>
</tr>
<tr>
<td>3.2.</td>
<td>Network graph</td>
<td>29</td>
</tr>
<tr>
<td>3.3.</td>
<td>Main Program</td>
<td>31</td>
</tr>
<tr>
<td>3.4.</td>
<td>Propagation rule 1</td>
<td>32</td>
</tr>
<tr>
<td>3.5.</td>
<td>Propagation rule 2</td>
<td>32</td>
</tr>
<tr>
<td>3.6.</td>
<td>Propagation rules 3, 4</td>
<td>33</td>
</tr>
<tr>
<td>3.7.</td>
<td>Marker Type classes 1, 2, 3, 4</td>
<td>33</td>
</tr>
<tr>
<td>3.8.</td>
<td>Memory Usage(k&gt;1)</td>
<td>38</td>
</tr>
</tbody>
</table>
4.1. Simple Connections (from [19]) ................................. 44
4.2. Gloss Connections (from [19]) ................................. 44
4.3. Method 1 (from [19]) ................................. 46
4.4. An example of filtering paths with Method 1 .......... 46
4.5. Text Inference Application Architecture ................. 48
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1. Dynamic Memory Usage</td>
<td>35</td>
</tr>
<tr>
<td>4.2. Propagation Relations (from [25])</td>
<td>43</td>
</tr>
<tr>
<td>4.3. Results for the Text Example</td>
<td>45</td>
</tr>
<tr>
<td>4.4. Examples of inference</td>
<td>47</td>
</tr>
<tr>
<td>4.5. Memory Usage without heuristic</td>
<td>52</td>
</tr>
<tr>
<td>4.6. Memory usage with the proposed heuristic</td>
<td>53</td>
</tr>
<tr>
<td>5.1. Examples of inference</td>
<td>55</td>
</tr>
</tbody>
</table>
To all those around me
CHAPTER 1
INTRODUCTION

1.1. Motivation

Computers are challenged with more and more complex problems. Text inference algorithm developed in [19] is highly parallel and requires distributed control. The conventional programming paradigms are not suitable for complex problems encountered in Artificial Intelligence (AI). To be able to encode those problems in a computing machinery we should design first powerful models that will allow us to express the computations of difficult applications at high level of abstraction. The implementation of these models has to be efficient. We need a powerful model with a good implementation.

The marker propagation model offers a high degree of parallelism and distributed control capabilities. It consists of a network of nodes (the web) in which a set of markers are propagated in parallel throughout the network. This model seems to be powerful enough to let us handle complex problems, like text inference [19]. Implementing a marker passer requires support for parallelism and lots of resources in terms of processors and memory that may be found in a distributed environment.

Object oriented view of the world is known as allowing for a more natural modelling of distributed and parallel systems [10]. This claim is based on the view of an object-oriented system as a collection of loosely-coupled objects communicating messages. From such an initial viewpoint, it is easy to perceive the objects in the system as separate parallel processes running on a single machine or even distributed over a network of machines. The high level programming style offered by object-oriented environments would be a good choice to handle the complexity of a marker
passer.

As a summary one may say that for the implementation of a marker propagation algorithm we need a programming environment which has:

- object-oriented support
- thread-level parallelism
- distributed capabilities

Java programming environment fits all these requirements.

In this thesis we present the design and implementation of a marker propagation framework for developing marker-propagation-based applications. We take advantage of Java threads to offer as much parallelism as possible. The distributed control is shared by different threads. The framework can be easily adapted to a specific application with a minimum programming effort: the only part which needs to be overridden is the implementation of propagation rules associated with a specific problem. All the mechanism of transmitting and receiving markers and their schedule at node level are handled by our framework.

1.2. General Goals

The main goal of this thesis is to prove that highly-parallel algorithms like the text inference presented in [19] are feasible and practical. A general framework for implementing marker passing based solutions to complex problems is developed. To testify its generality another application is developed, regarding the problem of pattern matching.

1.3. Thesis Outline

This thesis is organized in five chapters. Chapter 2 presents the Object-Oriented and marker propagation models and how to fit the latter into OO systems; it describes the design and implementation of the marker propagation framework in Java. Two
applications implemented in this framework are presented in Chapter 3 and 4. Specifically, we describe the implementation of a pattern matching in graphs (chapter 3) and a text inference application (Chapter 4). Finally, chapter 5 presents conclusions and possible future developments of our framework.
CHAPTER 2
OBJECT-ORIENTED MARKER PROPAGATION

The purpose of this chapter is to define and examine the concepts behind object-oriented and marker propagation models of computation. An initial discussion outlines basic concepts, definitions and objectives associated with object-oriented languages and systems. Then we describe the marker propagation model and its main features. Finally the two conceptual frameworks are joined together in order to examine how we can best develop a marker propagation system on top of a threaded object-oriented system. A possible Java implementation is presented after an overview of the Java environment.

2.1. Object-Oriented Systems

2.1.1. Definitions

As one might expect, the basic concept underlying object-oriented computing is that of an object. There have been several answers to the question What is an object?, due to the fact that the term emerged in several different fields of computing during the 70s. Typical notions of object that have emerged in various fields are:

1. Computational agents that carry out scripted actions in response to a message (from the areas of parallel computation and AI)

2. Packages of information and behavior within superclass/sub-class hierarchical structures (from Simula, Smalltalk and AI)

3. Abstract data types (from programming languages design)
4. Modules or units for knowledge and expertise, such as Minsky’s frame (from knowledge representation), and

5. Protected resources (from operating systems).

In all the cases, the primary goal behind these notions of object was to provide a means of managing software complexity. The common characteristics of these options are that an object is a self-contained entity that has a well-defined interface [42]. This is a rather vague definition. Other conceptual-level definitions have defined an object as a system entity[6] characterized by the set of actions that can be performed on it, or as any perceived entity within a software system. For the purpose of this thesis, a more concrete definition will be adopted. An object will be defined as follows:

Definition: An object is a software entity that packages together, or encapsulates, some set of private data and the set of operations, or methods, that can be externally invoked to access and manipulate that data [1] [10].

An object is an entity that maintains its own internal state. The object updates its state in accordance with the execution of its methods. The methods of an object are the only procedures that can access the object’s internal data, or state [10][42].
This concept of object encapsulation is illustrated in Figure 2.1.

Interaction among multiple objects is usually modeled as a flow of messages. When object A invokes method X of object B, object A is said to send a message X to object B.

It is important to realize, however, that this model of message passing is a conceptual model [1]. Many object-oriented systems do not send actual messages between objects [7]. Instead, the conceptual message is actually implemented as a form of procedure call[32]. This inconsistency between the conceptual model and the implementation creates a potential source of confusion. Object-oriented systems that support a true message passing implementation tend to provide natural support for parallelism and distribution[11][23]. Those systems that implement messages as either statically or dynamically bound procedure calls do not provide such support for parallelism and distribution [22](although their conceptual model implies that they do).

Another important concept underlying object-orientation is the idea of a class. Classes in object-oriented systems provide a means of classifying objects [10]. Different objects that belong to a common class share identical behavior. The class acts as a template that specifies the methods and private data elements for all objects that belong to the class [1]. Classes are used to create objects; every object is an instance of a class. Instantiation is the process of creating an object (instance) from a class.

Inheritance is another important object-oriented concept. Inheritance provides a means of easily reusing software by allowing new development to be based on existing code [10]. More specifically, inheritance is a mechanism that allows new classes of objects to be defined in terms of previously-defined classes by simply specifying how the new class differs from the original. The original classes are referred to as the base classes or superclasses of the new class. The new class is referred to as a derived class or a subclass of the original classes. A subclass inherits both data structures and methods from its superclasses; this eliminates the need to develop and maintain redundant code to implement common functionality. The class inheritance
relationships form a class hierarchy and more specialized classes are further down in the hierarchy.

The specialization of subclasses occurs in various ways [1]:

1. Addition of new behavior. This is the most common form of specialization. The subclass adds new functionality to the superclass in the form of new data and methods.

2. Change of existing behavior. This approach involves re-defining the behavior of an existing method. This is also a common way of achieving specialization.

3. Deletion of existing behavior. This type of specialization is less common. It involves removing data and methods from a superclass thus creating a subclass with less functionality.

Different object-oriented languages and systems provide various levels of support for inheritance. In fact, support for inheritance is sometimes used to classify programming languages. In this scheme, a language that supports objects but not inheritance is called an object-based language. Only languages that support both the concepts of object and inheritance are referred to as object-oriented under this classification. In this thesis, the term object-oriented is used to refer to a spectrum of support for object system concepts. This spectrum includes both object-based and object-oriented languages as defined above.

2.2. Marker Propagation Model

Quillian[38] has introduced the idea of marker-spreading activation as a reasoning mechanism, which evolved into the marker propagation model.

Marker propagation is a technique that was developed for using parallelism to find connections between concepts in semantic networks. The main components of a marker propagation model are: permanent knowledge, temporary knowledge and the inference mechanism.
- Permanent knowledge is a semantic network. Figure 2.2 shows a sample semantic network. The network nodes represent concepts or their properties, while the network arcs represent inter-relationships among the nodes. The network has a hierarchical structure in which more general or abstract nodes at the top subsume more specific nodes at the bottom. The inheritance property ordering saves memory because the properties of more general nodes are inherited by the more specific nodes.

- Temporary knowledge is represented as markers that are bit patterns attached to node data or more complex objects as regarded in this thesis. Thus, the temporary knowledge physically overlaps the permanent knowledge.

- Inference is achieved by changing the state of both temporary and permanent knowledge. Originally, markers are assigned to nodes by the global inference engine to indicate a specific knowledge state. Stimulated by the inference engine, markers move from node to node; while moving, they interact with selected markers in some other nodes along the path, changing the state of the knowledge base. The movement of markers is guided by propagation rules encoded and attached to them.

We can formalize the marker-propagation model as a triplet $MP = \{N, M, O\}$ where:
• N is the permanent knowledge or the semantic network. We may regard this permanent knowledge as not being static. That means some marker may decide to modify this network (see NETL’s cancellation links[21]).

• M is a set of markers which is dynamic by its nature. The markers are created, destroyed, replicated, scheduled, allocated.

• O is a set of operators that abstract the distributed control of the system. It contains operators on markers like create a new marker, replicate the marker(this is needed for the case when a marker has to travel along several links from a node; it makes a replica of itself for each new link in order to do the propagation along each link in parallel), destroy the marker(when a marker has no links to travel along from some node we need to destroy it) and also, a special operator that we call the scheduler, which influences the way in which markers (viewed as processing elements) are scheduled to execution. The scheduler determines the strategy adopted for exploring the semantic network (depth-first, breadth-first or others).

This model can be implemented in:

• **hardware** - this is the case of the SNAP project developed at USC by Moldovan et al.[40][39][8]

• **software** - and this is the approach which we consider in this thesis

The tradeoff between these two alternatives is speed over flexibility and costs.

A lot of work has been done in the field of marker propagation.

Quillian’s semantic memory model was developed to make a cognitive model of several memory processes concerned with word definitions and language tasks. His model kept word senses in an associative network form, and thus he was able to compare and contrast meanings by expanding out from each concept a set of activation tags. Each tag contained two components: the name of the original parent node at
which set of activations began, and a pointer to the immediate parent. As each new
node was activated it was checked to see if it had already held an activation tag. If it
had been activated from a different parent, then two paths were returned - the sets
of nodes tracing back to each of the parents. If it had been activated from the same
node as the present activation, then the system noted this and stopped tracing (thus
preventing loops). If the node had not been previously activated, it would be tagged
appropriately.

Fahlman[13] designed NETL to be a hardware implementation of marker-passing
scheme (actually simulated in software). The network memory was similar to that
of Quillian, but it was realized quite differently. Each node in memory was replaced
by a simple hardware device called a node unit, and each link became a hardware
link unit. These devices were able to propagate a set of markers in parallel, through-
out the network. The performance of these devices was controlled by a single serial
computer known as the network controller. NETL was designed to deal quickly with
certain types of knowledge-base issues: those concerning type hierarchies and prop-
erty inheritance. The system allowed two main kinds of nodes, individual-nodes and
type-nodes. Type nodes served as the templates for storing a set of properties about
some class of entities. The individual nodes were instances of some type allowing
inheritance. NETL works by passing markers throughout the system via the parallel
architecture of the various nodes and link processors. Essentially, this proceeds by
propagating several types of markers through the network; the controller processor is
able to query all nodes to find how they have been marked.

Hendler and Philips[21] developed a form of marker-passing in their PATI sys-
tem. A bidirectional breadth-first search methodology was used to create a conceptual
representation for abstracts from patents and to group them in a semantic network.
PATI was implemented using object-oriented features. The network was programmed
as a collection of objects of type node, each inheriting properties from the node class.
The bidirectional spreading activation mechanism was programmed as local decision
process.
A two-part marker-passer/screening mechanism was used in a model developed by Norvig. His system concentrated on the recovery of inferences from text. Norvig’s model had two-pass filter for determining whether a path was relevant. First, some paths were ruled out quickly because the pattern of links could not lead to a plausible inference. Second, a matching procedure was used to determine if the information in a path was truly relevant to the present story. The lengths of the paths were limited, so that this matching procedure could be kept efficient.

Massively parallel architectures have also been put to use in the attempt to do natural language processing. Moldovan[40] mapped the marker propagation model into a computer architecture consisting of a distributed intelligent network working under the supervision of a central controller(Figure 2.3). In [40][31] he used this architecture for knowledge processing on semantic networks. The intelligent network consists of an array of programmable 32-bit processors with each processor managing a collection of semantic networks nodes. Both the permanent knowledge and temporary knowledge are stored in the intelligent network. The role of the inference engine is shared by both the controller and the processing array. The controller performs global inferences and marker task initiations. But, since inferencing is achieved by moving markers in the semantic network, the bulk of the processing is performed by the intelligent network. This approach attempts to implement all the three elements of the marker model as close to hardware as possible[8][5].

2.3. Threaded Object-based marker propagation

All previous implementations of the marker propagation are either in hardware or in low level programming languages. This lacks flexibility and implies a considerable effort to develop a marker passer for a new problem. In this thesis we use the object-oriented technology to implement a framework for marker propagation that encapsulates the general issues of the marker model, leaving the specific duties to be implemented by the user of the framework.
We have developed an object-oriented design of the marker propagation model, enhanced with threads.

The permanent knowledge is mapped into a network of objects (Figure 2.4). Each node in the semantic network is stored into a concept object. The links are references from one object to another. The concept objects have methods to accept markers from the neighbor concepts and to allow them to access the information stored in it. Each concept object can be viewed as a host that may accept several visiting markers (Figure 2.5).

The temporary knowledge is mapped into marker objects which inherit the capabilities of a generic thread object. The Marker class, which abstracts the Marker objects, contains methods for the basic operations on markers: creation (the constructor), replication (the copy constructor), deletion (the destructor and garbage collector), ready to run (the start method associated with a processing element).
Inferences are obtained as a result of the processing performed by the propagation rules, executed as separated threads.

2.3.1. Semantic Network Abstraction

A node in the semantic network is an object that we call Host, which has associated with it a queue for markers. The incoming markers will be placed in the queue and started according to a particular local scheduling scheme (first-in-first-serve, priority-based algorithm or other). This local scheduling scheme will influence only the order in which the markers (actually the threads embedding the propagation rules of markers) are placed in the ready state and not yet in the run state. The latter transition is controlled by the system scheduler (either the one in the run-time or the one of the operating system if we use system-level threads). The default local scheduling scheme is a FIFO algorithm.

The structure of our generic Host class is presented in Figure 2.6.
**Host generic class**

Identity id;
Queue marker_queue;
LinksType links[];

//constructors
Host(Identity);
Host();

// copy constructors
Host(Host);

// queue processing procedures
void accept_Marker(Marker);
Marker next_to_run();
LinksType Look_up()
void add_link(Host);
void remove_link(Host);

Figure 2.6. Host generic object
As one may notice each host has a unique identifier hold in the \textbf{id} \ data member. The type of the \textbf{id} \ data member is a generic \textbf{Identity} type which can be chosen for each specific application (String, Integer). The queue may be implemented as a linked list or as a vector. The operation on the queue \textit{add-to-queue} and \textit{remove-from-queue} are implemented by the \textbf{accept-Marker} method, respectively \textbf{next-to-run}. In this latter method one may embed several scheduling schemes: first-in-first-out, priority-based (some markers are urgent, so that they will have to be started earlier) or other non-standard scheduling algorithms.

The method \textbf{lookup} is an abstract method; this means one can not have instances of the Host generic class. The user is enforced to rewrite this method by deriving a specific class for her application. \textbf{Lookup} will return the specific local information (like links to other hosts). The \textbf{links} array (which may be also a linked list) stores links to other hosts in the network. The linking information can be updated by using \textbf{addlink} and \textbf{removelink}.

A message schema of the generic Host object is illustrated in Figure 2.7.
2.3.2. The Markers

In our design, a marker is a threaded object. It has the properties of a thread, together with additional information that is used for marker passing. The main method of a marker is its propagation rule which constitutes the control flow of the thread. After a marker is created, by calling the constructor of the class, it will be started for execution on a host node. Depending on its propagation rule, a marker may stop its further propagation, it may spawn siblings or make other decisions. It may also report its status and synchronize with other markers. The generic class that embeds the general behavior of such an Marker object is the Marker class. A pseudo C++-like description of the class is shown next.

```cpp
class Marker extends Thread{
    Identity id;

    Host originalhost; // the host where the marker was created
    int replica; // which replica of a the original marker this marker is.

    Host current-host;
    // constructor
    Marker(Host);
    // copy constructor
    Marker(Marker);
    // propagation rule
    void run(); }
```

We assume that we have a predefined Thread class which encapsulates the behavior of a Thread.

```cpp
class Thread{
    // the code associated with this thread
    run()
    // it prepares the current thread for execution
    start()
}
```
The main method associated with a Thread object is `run` which contains its control flow. The `start()` method prepares the thread for execution. Our generic Marker class inherits the behavior of the Thread class, by deriving from it. We say that Thread is a superclass of Marker. All the work related to a thread object is hidden in the implementation of the Thread class.

A Marker object has an unique identification, provided by the `id` data member. We also keep track of the original Host object of the marker (the node where the marker started its journey). The third data member is an integer which stores a number that signifies what replica of the original marker the current marker object is. For example, in Figure 2.8 we can see a Host node which has three links. Let us assume the we have marker M running on that host. Its propagation rule imposes to propagate along each outer link from the current host.

The marker propagates itself along one link (the leftmost link) and spawn two replicas on the others. The replicas retain the original marker’s identity but they get an increased replica number. The replica number is valuable for path recovering and loop elimination.
The basic operations on marker objects are realized using the methods offered by the Marker class:

- **Creation.** A marker is created using the constructor method. A marker can be created in a propagation rule or in the main function.

- **Placement.** A variant of the `acceptMarker` method is used for placing a marker into an initial marker. It sets the value of the current host and starts the propagation rule of the marker. When a new marker is created its host is initialized to a void object and because the usual `acceptMarker` method requires a non-null current host, it cannot be used. Thus, we need a separate implementation of the placement operator.

- **Replication or Spawning.** For this, we utilize the copy constructor which is in charge of copying, member by member, the original marker into a new one. The only exception is the `replica` data member which must be incremented for each new copy of the original marker.

- **Firing.** That means activate the propagation rule of a marker. This is done by sending a message of type start to a marker object.

- **Propagate.** It refers to the main operation of marker passing: the passing of a marker from one concept (the source) to another (the destination), provided there is a semantic link between the source and the destination. The propagation is realized by sending an `acceptMarker` message from the source node to the destination node. During this operation the marker will be extracted from the marker queue of the source and placed in to the marker queue of the destination. The local scheduling operator influences the propagation of markers at host level, by determining the order in which the incoming markers are started.

- **Deletion.** A marker is killed when there are no other hosts to visit. This extermination may be done explicitly by using a destructor message or using an automated mechanism of garbage collection which is less error-prone.
A message schema of the Marker generic object is illustrated in Figure 2.9.
As a summary, the three elements of the marker propagation model (\(\{M, N, O\}\)) are mapped into the object model as described below:

- Semantic Network is a network of Host objects. The links are typed references from a source host to a destination host.
- Markers are threaded objects. Their propagation rules are embedded into threads.
- The set of operators is mapped into messages (methods) accepted by both Host objects and Marker objects.

### 2.4. Marker propagation in Java

In this section we present the design of the marker propagation model in Java Development Environment. We start by showing the capabilities of Java language and Java Virtual Machine and how we can benefit from them. After that we describe
the Java main classes we developed to build up a framework for marker passing similar to the one presented in the previous section.

2.4.1. Java Development Environment

Java\(^{17}\) is a general-purpose concurrent object-oriented programming language. Its syntax is similar to C and C++, but it omits many of the features that make C and C++ complex, confusing and unsafe. It was designed to support multiple host architectures and to allow secure delivery of software components. Compiled Java code had to survive transport across networks, operate on any client, and assure the client it was safe to run. Moreover, Java has built-in support for multithreading.

Java’s main attraction is its portability. Rather than producing machine-specific instructions, the Java compiler produces platform-neutral bytecode. The Java run time environment’s interpreter translates the bytecode into actual machine-specific instructions.

Java Development Environment\(^{33}\) is together a language and a virtual machine which constitutes its run time system. It differs from others by the specific design of the run time system, merely a virtual machine with its own instruction set called Java bytecode or simple bytecode. The Java computing model is shown in Figure 2.10.

2.4.1.1. Java - the language

Java\(^{24}|^{26}\) contains common numeric datatypes as short, int, long float, and double. Java data types, unlike some implementations of C, are independent of the underlying hardware and operating system. Java has a true character type (16-bit unicode characters). Java arrays are true objects. All arrays have a length associated with them, checked both at compile and run time. In Java, all variables are strongly typed.
Figure 2.10. Java computing model
2.4.1.2. Java Performance

We may reach a factor of 2-4 lower performance from a PC JIT compiler compared to C compiled code [30][2]. For the portable applet mode this is as big as a factor of two[12][4]. However with some restrictions on programming style[15], we expect Java language and VM compilers to be competitive with the best Fortran and C compilers [9].

2.4.2. Java Threads

Threads are created and managed by the classes Thread and ThreadGroup. Creating a Thread object creates a thread. When the thread is created, it is not yet active; it begins to run when its start method is called.

2.4.2.1. Synchronization

The Java language does not provide a way to perform separate lock and unlock operations; instead, they are implicitly performed by high-level constructs that arrange always to pair such operations constructs. The Java Virtual Machine, provides separate monitorenter and monitorexit instructions that implement the lock and unlock operations. There is a lock associated with every object. The synchronized statement computes a reference to an object; it then attempts to perform a lock operation on that object and does not proceed further until the lock operation has successfully completed. (A lock operation may be delayed because the rules regarding locks can prevent the main memory from participating until some other thread is ready to perform one or more unlock operations.) After the lock operation has been performed, the body of the synchronized statement is executed. If the execution of the body is ever completed, an unlock operation is automatically performed on that same lock. A synchronized method automatically performs a lock operation when it is invoked; its body is not executed until the lock operation has been successfully completed. If the method is an instance method, it locks the lock associated with the instance for which it was invoked (that is, the object that will be known as this

22
during the execution of the body of the method). If the method is static, it locks the
lock associated with the Class object that represents the class in which the method
is defined. Again if the execution of the method’s body is ever completed, an unlock
operation is performed on that lock. If a variable is ever to be assigned by one thread
and used and assigned by another, then all accesses to that variable should be enclosed
in synchronized methods or synchronized statements. Every object, in addition to
having an associated lock, has an associated wait set, which is a set of threads. When
an object is first created, its wait set is empty. Wait sets are used by the methods
wait, notify, and notifyAll of class Object. These methods also interact with the
scheduling mechanism for threads. The method wait should be invoked for an object
only when the current thread (let us denote it with T) has already locked the object’s
lock. The wait method then adds the current thread to the wait set for the object,
disables the current thread for thread scheduling purposes, and performs an unlock.
The thread T then lies dormant until one of the following events happens:

- Some other thread invokes the notify method for that object, and thread T
  happens to be the one arbitrarily chosen as the one to be notified.

- Some other thread invokes the notifyAll method for that object.

- If the call by the thread T to the wait method specified a time-out interval,
  and the specified amount of real time has elapsed.

The thread T is then removed from the wait set and re-enabled for thread scheduling.
The notify method should be invoked for an object only when the current thread has
already locked the object’s lock, or an IllegalMonitorStateException will be thrown. If
the wait set for the object is not empty, then some arbitrarily chosen thread is removed
from the wait set and re-enabled for thread scheduling. (the thread will not be able
to proceed until the current thread relinquishes the object’s lock). The notifyAll
method has already locked the object’s lock, or an IllegalMonitorStateException will
be thrown. Every thread in the wait set for the object is removed from the wait set
and re-enabled for thread scheduling (those threads will not be able to proceed until the current thread relinquishes the object’s lock).

2.4.2.2. The Thread class

Java contains a powerful, simple Thread object that one may instantiate in his program. The Thread class provides all the functionality needed for you to create fully multithreaded, safe applications. There are two ways for spawning threads in Java:

- using the entire class as a thread
- inheriting from the Thread class

The first approach creates a runnable class and attaches to a thread. The entire class exists within the thread and the stream of execution for that class is maintained by the thread. If the thread is killed, the stream of execution is destroyed also. The biggest advantage of this method is that the class does not need to know anything about how it is to be implemented.

The second way to implement threads is to create a class that inherits from the Thread class. In the first approach, we have created an object on its own. In this second case we create a Thread object from the beginning. The thread code for a class that inherits from Thread is in the run method. Any code needed to manage the thread should be placed within this method. It is also in this method the place where a thread can be suspended or sent to sleep. The difference between extending the Thread class and implementing the Runnable interface is that when you inherit from Thread, your entire class is a thread. The thread must be started and stopped from within the class, unlike the other method in which the thread controls are outside the class itself.

We prefer inheriting from the Thread class because it leads to a more flexible control of a thread.
2.4.2.3. Thread controls

A Thread has several control methods that affect its behavior. In order to have complete control of a thread Java provides the following methods: Start, Stop, Suspend, Resume and Sleep.

The start method tells the thread that it may begin the execution of all the steps contained in the run method. The stop routine terminates the thread and prevents the run method from executing any further steps. It does not kill any child threads that it may have created. When suspend is called, the thread stops the execution of its run method until resume is called somewhere down the line. A parent may get information about the current running status of its children by calling isAlive and find out if the thread is stopped. If the thread it is not running, and it is not stopped, it must be suspended. The sleep method tells the thread to pause for a given number of milliseconds.
2.4.3. Marker Propagation Implementation in Java

We abstracted the behavior of a marker model and designed an object-oriented model that contains elements needed in every implementation of marker-based solutions to problems. We considered the concepts in the semantic network as host elements whose behavior is to accept markers and provide local data to the propagation rules of the accepted markers. The local context (the semantic information) is specific to an application. The user is the one that should specialize the abstract host node and specialize the local context. The markers in our design are light-weighted processes, namely threads. Their main control flow will implement the propagation rule attached to the marker. A marker has an unique ID (signature) and a type. Usually, a marker type has one propagation rule but this is not always the case. The propagation rules process the information at the host and takes several decisions: replicate himself, commit suicide, report to the another thread its status, spawn other threads.

The life-time cycle of a marker is: the marker is created and placed into an original host by the main application (or network controller). Based on the particular needs of the client application, a marker has its own propagation rule. Once accepted by a host, the marker gets its own thread of execution in which to perform its rule. In many cases the marker queries the host by calling the look-up function. When an marker is ready to be transferred, its destination host is asked by sending an acceptHost() message to it: newHost.acceptMarker(), which passes the marker’s identity to be sent along. The destination then “pulls” the marker from the sender by calling currentHost.extractMarker() which returns the Marker object. A separate thread is created for the marker and started. This is the default first-in-first-come local scheduling scheme. The marker, the thread and a monitor that waits for the thread to finish are stored in arrays using the marker identity as a key. When the marker’s method is finished the monitor calls new Host.acceptMarker() and the marker goes along. The Java implementation of this system implements the abstract class Host using interfaces and the Marker object by deriving it from the Thread
class offered by Java standard class library. The generic Host object described in the previous chapter is implemented using an interface and a class that implements the interface. We used the interface because of future development reasons (distribution aspects). The generic Marker is implemented as a subclass of the predefined Thread class. The generic interfaces of these two classes have been presented in the previous section when we described the generic Marker and Host classes from an Object-Oriented point of view. They have an one-to-one correspondance in Java. A marker manager and a Host manager may be required by particular applications, but this is not always the case. See for example the pattern matching applications, where there is no such a need. All those new classes are serializable (an important feature of Java that allows objects to be send along a stream, namely a network stream) which allows for distributed extensions. The markers on hosts will be scheduled by the Java Run Time scheduler in a round-robin fashion. At host level, the markers are started in a first-in-first-out manner.

2.5. Summary

In this chapter, it was described an object oriented approach to marker propagation. The object oriented and marker propagation models were presented and then a possible mapping of the latter onto the former was detailed. This mapping is general and constitutes as an infrastructure for marker-propagation based implementations. An implementation of this mapping was thereafter explained. The Java development environment was choosen for such an implementation because of its object-oriented, distributed and portability features. In the next two chapters we develop two application in order to prove the strengths and weaknesses of this approach.
CHAPTER 3
PATTERN MATCHING APPLICATION

In this chapter we present a pattern matching application in which we have to search for a graph (pattern) in a larger network. For a length of the critical path $P$, the matchings are found in a time linear with $P$. After the problem is presented, we describe how to adapt our marker propagation framework, developed in Java, to this particular problem. There are different markers with different propagation rules. The pattern to find is encoded in the paths that have to be followed by different types of markers and in the synchronization points among those markers. The application is tested on a graph and the results (the matchings found) are displayed. For big networks that do not fit in memory, there may be the case of not being able to find any solutions because of missing nodes from memory. We propose a heuristic that allows a 33% percent increase for the number of nodes loaded in memory.

3.1. Application description

The application we are trying to solve performs a pattern matching of the graph illustrated in Figure 3.1 (a) to the general graph illustrated in Figure 3.2 (b), that is we try to determine if network in (b) contains the pattern (a). The input variables are the nodes X1 and X17, while the output variables are the possible bindings that may exist for this set of nodes. If there are no bindings, then the input pattern (a) cannot be found in the network (b), but if there are bindings, each binding set represents a solution.

The mapping of this pattern matching problem into a Marker Propagation Network is straightforward. To describe the implementation, we will use the SNAP[41] instruction set which seems to be quite useful, clear and clean [40].

28
Figure 3.1. Pattern

Figure 3.2. Network graph
The **MPN_Pattern_Matching** program in Figure 3.3 creates markers m1 and m2 in all nodes. These markers have propagation rule 1 and propagation rule 2, respectively. The idea is to send markers over the network such that the propagation rules will spawn markers when the path splits and collide markers when the paths merge. These partial paths are the loops of the pattern. Finally, a marker m3 results if all conditions imposed by the pattern were satisfied when markers propagated in the knowledge base. For this problem, seven different markers were necessary, each marker with a different propagation rule that reflect the pattern. The propagation rule of m1 creates marker m3 whereas rule 2 generates marker m4 and m5, that propagate according to rule 3 and rule4 respectively. Finally, rule 4 generates marker m6 and m7, propagating according to rule 5 and rule 6 respectively. In addition to its propagation rule, each marker has two mandatory fields: the marker type and an array of integers indicating the indexes of the nodes from the network where the pattern matched. To recognize the nodes that play the role of X1 and X2 in the pattern, we propagate m1 from X1 along an R1 relation and concurrently marker m2 from X2 along an R3 relation. At its destination, m2 creates two other markers, m4 and m5. Marker m4 will continue to propagate along the sequence of relations R4, R2 and R2 (recognizing nodes for X9, X8 and X7) and will synchronize with m1 at the node where m2 started to propagate (playing the role of X2). After m1 and m4 synchronize (illustrated in rule1 by the primitive cond(4)), marker m1 continues to propagate along relations R1 and R3 (collecting the indexes of the nodes playing the roles of X3 and X4). At that point, m1 must synchronize with marker m5. In the same time marker m5 (created at X9 by m2) propagates along R4 and creates at the destination (X10) two other markers m6 and m7. Marker m6 propagates along the path X10-R4-X12-R3-X4 and permits m1 to further propagate along R4, thus reaching X5, where it synchronizes with m7. Marker m7 propagates concurrently along the path X10-R4-X11-R1-X13-R1-X14-R3-X5. Finally, after the synchronization between m1 and m7 at X5, marker m1 propagates along R2 to recognize the node having the role of X6, thus completing the pattern matching and creating marker m3 that is going to be collected and send
to the host, as solution of the problem. Whenever one of the seven markers used in this solution fail to propagate, because either there is no relation along which they can propagate or they cannot synchronize with other markers, this indicates that the pattern match could not be produced. For this particular example, the initial 42 pairs of markers m1 and m2 return only 5 solutions.

This implementation is straightforward. We found and implemented an improvement to this which reduces the number of types of markers. There are only four types of markers now: 1, 2, 3 and 4. The main program starts by loading the network of nodes into main memory. It then places a marker of type 1 in each node. Marker type 1 first propagates along a R1 relation from its original host. If many such relations exist the marker is replicated. The replicas retain the original’s id and have a counter that distinguish among replicas. After this first step, 1 waits for a type 2 marker to come in. It may be the case that no one arrives so that 1 will be stopped here. At the end of the application or after a while (we may set a timer) this thread should be killed. In the case that the synchronization point is satisfied 1 propagates along an R1 and further along an R3 relation. Again a synchronization point is encountered. This time 1 waits for a type 3 marker to come along (created by a type 2 marker that was originally in the second host visited by marker 1). Then, it goes along a R4 relation and waits for a type 4 marker. The last step is to go along a R2 relation. If the marker reaches this point a match can be reported. The code for the propagation
Figure 3.4. Propagation rule 1

Propagation Rule 1
{ Marker2 m2;
  seq(m1,R1); create(M2); cond(m2); update_match();
  seq(m2, R1, R3); cond(m3); update_match();
  seq(m1, R4); cond(m4); update_match();
  seq(m1, R2);
  print_match()
}

Figure 3.5. Propagation rule 2

Propagation Rule 2
{
  Marker m3;
  seq(m2, R3); create(m3);
  seq(m1, R4, R2, R2, R2);
}

rule of marker type 1 is shown next.

Marker type 2 only propagates along a R3, R4, R2, R2 chain of relations and after the first step it sends a marker of type 3 onto the network. It also checks that it arrives at the same host it started.

Marker 3 travels along R4, R4, R3 and it creates a type 4 marker after the first step. Marker 4 goes along R4, R1, R1, R3. The propagation rules 2, 3 and 4 are given in Figure 3.5, respectively 3.6.

3.2. Implementation and Results

In the implementation, we specialized the Marker class for each of the four marker types defined in the previous section. Thus, we have the following classes Marker1 which implements the propagation rule 1, Marker2 which implements
propagation rule 2, Marker3 which implements propagation rule 3 and Marker4 which implements propagation rule 4. All these four specializations extend the generic class Marker and the only task the programmer has to do is to write the run method of these classes which should be the propagation rule associated with the type of marker it belongs to. There is one more thing that the specialized classes have, compared to the generic class: the where_to_go array which is an array of short numbers that encodes the path (sequence of typed links) a marker should follow. This can be viewed as a set of constraints which restrict the number of paths a marker may go.
If we run the program for the pattern in Figure 3.1 and the network in Figure 3.2 we find five matchings:

\[
\begin{align*}
X1 &= 14, X2 = 8, X3 = 15, X4 = 16, X5 = 17, X6 = 27, X7 = 7, X8 = 2 \\
X9 &= 3, X10 = 4, X11 = 10, X12 = 9, X13 = 5, X14 = 11
\end{align*}
\]

\[
\begin{align*}
X1 &= 14, X2 = 8, X3 = 15, X4 = 16, X5 = 17, X6 = 18, X7 = 7, X8 = 2 \\
X9 &= 3, X10 = 4, X11 = 11, X12 = 9, X13 = 5, X14 = 11
\end{align*}
\]

\[
\begin{align*}
X1 &= 14, X2 = 8, X3 = 9, X4 = 16, X5 = 17, X6 = 27, X7 = 7, X8 = 2 \\
X9 &= 3, X10 = 4, X11 = 10, X12 = 9, X13 = 5, X14 = 11
\end{align*}
\]

\[
\begin{align*}
X1 &= 20, X2 = 22, X3 = 25, X4 = 26, X5 = 28, X6 = 29, X7 = 31, X8 = 32 \\
X9 &= 23, X10 = 24, X11 = 33, X12 = 34, X13 = 39, X14 = 40
\end{align*}
\]

The search is **exhaustive**. We ran the application using only markers of type 1 and 2 (that is a subpattern of the original one) with no synchronization and we got only five path matchings which coincide with the above ones for the first 9 bindings (the subpattern has only nine entries). The time complexity is \(O(P)\) where \(P\) is the size length of the critical path in the pattern. Each marker propagation from one node to another counts for one step. M1 visits 6 nodes, M2, M3 and M4 visit each 4 nodes. Some of the steps are executed in parallel (M2, M3 and M4). The total number of steps is actually 8. For the worst case when there is no parallelism among markers the same solution can be provided after a number of \(S\) steps, where \(S\) is the size of the pattern. We performed several tests to see how the net size would affect the memory usage. This time complexity expresses the available parallelism that may be exploited when we have enough processors for all the thread. When only one CPU
Table 3.1. Dynamic Memory Usage

<table>
<thead>
<tr>
<th>Propagation wave</th>
<th>Memory usage (Network and Markers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>840</td>
</tr>
<tr>
<td>1</td>
<td>578</td>
</tr>
<tr>
<td>2</td>
<td>394</td>
</tr>
</tbody>
</table>

is available we do not have any significant improvement over the serial case. We made the following study. We increased the network 15 times and tried to run the application. It ran out of memory (we had 630 nodes for the net plus 630 type 1 markers). We tried to increase the net 10 times (420 nodes plus 420 type 1 markers) and then measure the dynamic evolution of memory usage. The network would be the same but the markers will be dying rapidly.

As one may notice the memory usage tends to the size of the loaded network. This is due to the fact that once a marker finishes his path-to-go it dies (either reporting a matching or just dissapearing). It is interesting to observe that after few propagation steps the number of active markers is reduced with a factor of $k$. We detected the moment when half of the active markers are not active anymore. At this point, we forced the garbage collector to clean the memory and we started loading another portion of the data. We loaded as many new nodes as half of the in-memory ones (the number of initial markers equals the net size in memory). That is a quarter of the size of the network initially loaded in memory. Thus, we can repeat this process and get an increasing in-memory network. It can be modeled by the series:

$$M + \frac{1}{4}M + \frac{1}{16}M + ... + \frac{1}{4^n}M \quad (3.1)$$

if $n$ is big enough ($\infty$) we have:
\[ \sum_{n=0}^{\infty} \frac{1}{4^n} M = \lim_{n \to \infty} \frac{1 - \frac{1}{4^n}}{1 - \frac{1}{2^n}} M \]  

(3.2)

and we obtain

\[ \lim_{n \to \infty} \frac{1 - \frac{1}{4^n}}{1 - \frac{1}{2^n}} M = \frac{4}{3} M \]  

(3.3)

Which is an improvement of 33% of the size of the net in memory upon which we search for matchings. However, if we monitor the moment when all the markers are not active anymore, we may reuse their memory space by loading half of it with new network concepts and half of it with type 1 markers. For this new net we have the series:

\[ M + \frac{1}{2} M + \frac{1}{2} M + ... + \frac{1}{2^n} M \]  

(3.4)

if \( n \) is big enough(\( \infty \)) we get:

\[ \sum_{n=0}^{\infty} \frac{1}{2^n} M = \lim_{n \to \infty} \frac{1 - \frac{1}{2^n}}{1 - \frac{1}{2}} M \]  

(3.5)

and further:

\[ \lim_{n \to \infty} \frac{1 - \frac{1}{2^n}}{1 - \frac{1}{2}} M = 2M \]  

(3.6)

An intuitive drawing of our formal analysis of memory usage is illustrated Figure 3.8. In this latter case we obtain an 100% improvement in memory usage with respect to the base in memory. An intuitive drawing of this is illustrated in Figure 3.8. We may generalize it as a function of the marker decaying factor. The marker decaying
factor is the fraction of the original number of markers and the number of active markers at some later point in time. As one may notice $k > 1$ for pattern matching. The range in which $k$ varies may be controlled by the programmer. For example, in the previous analysis, we chose to make a decision of loading more concepts when we have half of the initial markers ($k = 2$), respectively, when all the markers are not active ($k = 1$) anymore. This improvement in the semantic network loaded in memory (of up to 100%) is possible because in the case of pattern matching application the value of $k$ is larger than 1. In the pattern matching problem $k$ is larger than 1 because of the many constraints encoded in the propagation rules. For other applications where $k$ is smaller than 1 (as we will see in the next chapter) the active markers will dominate the memory usage and some heuristics to balance memory resources between the permanent knowledge and the temporary knowledge. Using this dynamic loading of the semantic network we may resolve problems of larger size, although the improvement is limited by the physical resources of the machine (we may load a semantic network as large as the machine’s memory permits, but we will not be able to do any processing because there will not be space for markers, the temporary knowledge).
Figure 3.8. Memory Usage (k > 1)
CHAPTER 4
TEXT INFERENCE

In this chapter we present an application related to natural language processing. We use an algorithm for text inference proposed in [19]. The algorithm offers a possible solution for the text inference problem; it extracts information unstated in the text but implied, using world knowledge. The idea of text inference is to find semantic paths among concepts. The algorithm is highly parallel. It uses WordNet, a semantic English dictionary available in electronic form, as world knowledge. The semantic network, namely WordNet, is so big that it would be impossible to load it into memory ahead of time. We propose and apply a lazy loading of the nodes in memory. On the other hand, in this application the marker spawning dominates the memory resources, as opposed to the pattern matching problem; to handle this we propose a heuristic to reduce the number of markers generated, by limiting the spawning to only a subset of the links starting from a host concept. Once we obtained the paths among concepts, the most important issue is to find out the relevant ones. We apply one of the methods developed in [19] to filter the paths. Those methods retain only the paths that explain the lexical relations among concepts in the original text. It is shown how the algorithm runs on a piece of text and what meaningful inferences it may extract.

4.1. The problem

Text Inference refers to the problem of extracting relevant, unstated information from a text. The implied information contributes to the understanding of the context of the text and also accounts for text coherence. Inference from text or speech
are powered by the world knowledge available. If we consider the text as presented in [20]:

\[ S1: \text{John hit the ball with the bat.} \]

\[ S2: \text{It landed far away.} \]

A system is expected to infer that John was playing baseball, that he is the batter, that hitting the ball causes the ball to move and that John must have hit the ball hard since it landed far away. The approach presented in [19] is searching for the semantic connections in the knowledge base that correspond to the lexical relations between words in the text. The knowledge base search aims to find a local context for each sentence, and furthermore to establish intersections or connections between these local contexts. Thus, for a given text, the algorithm finds a web of knowledge base concepts that semantically relate to other words in the text. Inferences result as an English interpretation of the context web.[29][27]. Massive parallelism is achieved by allowing parallel searches on the knowledge base.

4.1.1. The Knowledge Base

The knowledge base used in [19] is WordNet1.5, a lexical database developed at Princeton.

WordNet[35][25] was built as a semantic dictionary, in which words are searched based on conceptual affinity with other words. It covers the vast majority of nouns, verbs, adjectives and adverbs from the English language. The words in WordNet are organized in synonym sets[37], called synsets. Each synset represents a concept. There is a reach set of relation links between words and other words, between words and synsets and between synsets. A large linguistic knowledge base may be developed starting from WordNet[14][36].

All the words with the same meaning are connected to a concept node representing that meaning. A word may belong to several synsets that correspond to the various meanings of that word. The noun and verb concepts are structured into hierarchies by using ordering relations such as isa, is-part or is-member. The ad-
Table 4.1. WordNet overview [35]

<table>
<thead>
<tr>
<th>Part of speech</th>
<th>words</th>
<th>concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>noun</td>
<td>107,484</td>
<td>60,560</td>
</tr>
<tr>
<td>verb</td>
<td>25,768</td>
<td>11,364</td>
</tr>
<tr>
<td>adjective</td>
<td>28,762</td>
<td>16,428</td>
</tr>
<tr>
<td>adverb</td>
<td>6,203</td>
<td>3,243</td>
</tr>
<tr>
<td>Total</td>
<td>168,217</td>
<td>91,595</td>
</tr>
</tbody>
</table>

Adjectives and adverbs don’t have hierarchies, instead they are linked into clusters. Verb concepts are connected by semantic relations reflecting common-sense causality: the entail and cause-to relations. In WordNet, typical features of nouns are represented by attribute relations, connecting noun concepts to adjective concepts. Certain relations connect words to the nouns, describing typical situations in which their use is accountable.

Another characteristic of WordNet is that almost all the represented concepts have textual glossaries attached to them. Glosses contain representative information about the concepts, defining typical microcontexts in which the contexts operate. The glosses make explicit the functional relations to other concepts from WordNet, enhancing the degree of connectivity by an order of magnitude. In [18] it is described an Extended WordNet in which each concept’s gloss is transformed into a defining feature directed acyclic graph with nodes and links. This allows to find semantic paths between pairs of concepts by integrating various types of semantic information readily available in WordNet. Table 4.1 shows the number of words and concepts for each part of speech. In WordNet 1.5 there are 168,217 words organized in 91,595 synsets, thus a total of 259,812 nodes.
4.1.2. Algorithm Assumptions and Approach

The input of the algorithm is a text lexically tagged, semantically disambiguated, and spanned by lexical relations. The text is then transformed into a forest of text graphs. To each sentence it corresponds a graph with the nodes being the sentence concepts, while the edges are the lexical relations. The inference algorithm generates new information by identifying knowledge base concepts that relate semantically with the text. A novelty of this approach is that it tries to identify knowledge base paths that explain the textual lexical relations. If we take the relation \textit{object}(hit, ball) from the text example given, the algorithm searches for all the knowledge base paths that explain the \textit{object} relation between the words hit and ball. After such paths are found for each lexical relation, it is possible to unify the paths into a graph that consolidates the context of each sentence. This way, the semantic connections in the knowledge base are strengthen and reinforced by the intersection of the paths. A context for the sentence is established in the knowledge base as the totality of concepts along the semantic connections that correspond to the sentence lexical relations. The rich information provided collectively by the text input is used to filter out irrelevant semantic connections. The next level is to take advantage that the text is assumed coherent, meaning that sentences follow logically, one after another. This means that we can look for concepts that are common to two or more sentence contexts. In addition to these common concepts, we may search in the knowledge base for simple connections that establish links between various sentence contexts. At the end, inferences are extracted from each connection.

4.1.3. Parallel Search Engine

The search engine is considered to be responsible for finding semantic connections between pairs of knowledge base concepts. It implements three primitives search procedures that are used a lot: \textit{simple-connection}, \textit{gloss-connection} and \textit{combine-connection}. Each primitive constructs a semantic path by using a different set of knowledge base relations. The \textit{simple-connection} primitive uses only
node relations, the **gloss-connection** primitive uses only gloss relations, and the **combine-connection** uses both node and gloss relations. A common characteristic is that the search originates from both ends in parallel until all common concepts are found. Thus, a path is formed from two halves, one originating from one concept and the other from the other concept.

The procedure **simple-connection** starts the search from two concepts that have the same part of speech. For originating concepts, as well as for every concept visited by the procedure, the set of node relations along which the search may proceed is given in Table 4.2.

The search is successful, meaning that a simple connection is established, whenever a concept from the knowledge base is found such that it is connected to both originating concepts through a chain of simple relations. The simple connection is the juxtaposition of the two chains of relations and concepts. Figure 4.1 illustrates how this procedure works.

The basic procedure establishes connections between two concepts by using only gloss relations. The rationale is that since glosses provide definitions of concepts, it may be possible to link semantically two concepts by identifying the common features that describe them. The depth of the search or the number of glosses visited is arbitrarily set. Figure 4.2 gives an example of a path realized by this procedure.
Figure 4.1. Simple Connections (from [19])

Figure 4.2. Gloss Connections (from [19])
Table 4.3. Results for the Text Example

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Total paths</th>
<th>MSC paths</th>
<th>Method 1 paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

The third basic procedure, combined-connection combines, in parallel, the two previous search methods by using both the link lists from the first method and the lexical gloss relations from the second. Its level of parallelism sums those of the previously defined basic functions.

4.2. An example

We will consider as input the following text:

_In his first major management move since taking the helm of the Securities and Exchange Commission, Chairman Arthur Levitt consolidated control of the SEC’s 12 regional and branch offices under five regional directors._

_All five regional directors will report to William McLucas, head of the SEC’s enforcement division in Washington._

We tagged the text[3], disambiguated[34] and then found the lexical relations among basic concepts and provide our system with 17 concepts for the first sentence, respectively 8 for the second one. The results are displayed in the first row of Table 4.3.

In the second from left column we have the total number of collisions we found. We consider this result as being a good one if we take in consideration that the search was performed only along isa relations and few glosses.

The second column lists the number of effective collisions found. That means we only consider here the most specific (first) collision between two concepts at the
most specific common concept (from this point on there will be many collisions, all the way up in the knowledge hierarchy). Finally, the last column displays the number of paths that explain the lexical relations in the original text. This number was found by filtering the resulting paths using Method 1 from [19]. This shows how efficient this method is: it only provides three valuable paths that we may use to obtain inferences.

Examples of inferences are presented in Table 4.4.
Table 4.4. Examples of inference

<table>
<thead>
<tr>
<th>Chairman ... taking helm</th>
<th>takegloss assume offices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chairman ... assuming office</td>
<td>position isa office</td>
</tr>
<tr>
<td>Chairman ... assuming position</td>
<td>helm gloss position</td>
</tr>
<tr>
<td>Chairman ... assuming helm</td>
<td>textual concept reached. Stop.</td>
</tr>
<tr>
<td>all directors will report</td>
<td>director isa ... isa leader</td>
</tr>
<tr>
<td>all leaders will report</td>
<td>leader gloss inspire</td>
</tr>
<tr>
<td>all leaders inspire</td>
<td>inspiration pertaining inspire</td>
</tr>
<tr>
<td>all leaders cause inspiration</td>
<td>inspiration gloss mind gloss ideas</td>
</tr>
<tr>
<td>all leaders cause ideas</td>
<td>communicate gloss ideas</td>
</tr>
<tr>
<td>all leaders communicate</td>
<td>inform isa communicate</td>
</tr>
<tr>
<td>all leaders inform</td>
<td>report isa inform</td>
</tr>
<tr>
<td>all leaders report</td>
<td>textual concept reached. Stop.</td>
</tr>
</tbody>
</table>

4.3. Design Issues

The architecture of the solution for the text inference problem is illustrated in the Figure 4.5.

The WordNet Server is a multiclient server that waits for requests from clients. Every time a new connection is established a new thread is spawned to deal with the new client. What is interesting is that when each client creates a new object of type Concept, it is creating a thread that talks to the server, which in turn queries the WordNet to get the desired data.

One of the main issues in designing our server was the design of the interface between Java and WordNet. The WordNet library is in C. To get the information returned by WordNet we may either use Java Native Interface[16] capabilities or a more straight approach of using a stream connection. Java Native Interface is hard to use from a programmer point of view, especially on the debugging phase. Moreover, it seems that it is not well developed and this gives an hazardous source of errors. We chose the stream approach which gives us the opportunity to use a simple design.
based on stream classes of the Java environment. The main stream is not a raw
stream of bytes, but one that handles objects. A multiclient server was implemented.
The big advantage is obvious: many clients served at the same time; thus, increasing
the throughput of the WordNet server.

The server is very simple. We have a WNMServer class that does nothing but
listening to the port and whenever a new connection is requested, it creates an object
of type WNServerThread that will communicate with the new client.

The class WNServerThread after it opened two object streams, one for receiving
words and sense numbers and one for sending back the Concept objects, has three
functions: to read the incoming words and sense number, query the WordNet about
the incoming word on the input object stream and reply with an Concept object on
the output stream.

With this implementation the users are provided with an interface to WordNet
similar to the one defined in the C library. The only difference is that you get back
a Concept object instead of a Synset structure.

Once the information is retrieved from WordNet, it is stored in a Concept object
which inherits from the generic class MarkerHost. The main data fields are presented
next.

    public class Concept extends MarkerHost
    {
        long hereiam; // current file position
        int sttype; // type of ADJ synset
        int fnum; // file number that synset comes from
        short npos; // numeric pos  1 = NOUN, ... (this field is added by us)
        String pos; // part of speech
        int wcount; // number of words in synset
        String[] words; // words in synset
        int[] lexid; // unique id in lexicographer file
        int[] wnsns; // sense number in WordNet
        int whichword; // which word in synset we are looking for
        int ptrcount; // number of pointers
        int[] ptrtyp; // pointer types
        long[] ptroff; // pointer offsets
        int[] ppos; // pointer part of speech
        int[] pto; // pointer 'to' fields
        int[] pfrm; // pointer 'from' fields
        int fcount; // number of verb frames
        int[] frmid; // frame numbers
        int[] frpto; // frame 'to' fields
        String defn; // synset gloss(definition)
        public long[] gl; // gloss concepts
        public short[] gll; // gloss’s lexical relations
        public short[] gl-pos; // their pos
public int ngl = 0;

}

This structure is almost the same compared to the original WordNet structure. We added the field npos (numeric part of speech) which stores the part of speech (1 = NOUN, 2 = VERB, 3 = ADJECTIVE, 4 = ADVERB) in numeric form. We use only a part of these fields in order to reduce the memory costs of a concept. We keep the (hereiam, pos) fields, used to uniquely identify a concept in memory, and the pointers. As soon as a new Concept is reached along a link, if it is not yet in memory it will be retrieved from the lexical base and registered with the Concept Manager, which is actually a hashtable (it allows for fast retrieval). In the case the Concept is already active in memory (that means it was already reached along some other paths) it will be found by the Concept Manager and a reference will be returned. The concepts are uniquely identified in memory by the key (hereiam, pos). The hereiam field only cannot be used as a key, because there might happen that two different concepts have the same offset value(hereiam), given the WordNet organization of different part of speech in different files.

The concepts are activated in memory lazily. A concept would be retrieved from the lexical database only when it is reached along a link from an already active concept.

The execution flow of the application is:

1. Load in the original concepts. This means we have to provide the basic concepts of the original text.

2. Place a marker in each original concept by sending an acceptMarker to each original concept. An original concept is also a MarkerHost object that may accept such a message (see Chapter 2)). Once in a Concept(MarkerHost), a marker’s run method (its propagation rule) is started.

3. Execute the propagation rule. It first looks up in the visitors field of the concept and checks for collisions with other markers. A cycle test is done to be
sure that the markers have not yet visited that host. If this is the case it stops and reports the reason of quitting the propagation. Otherwise, it looks up in the host concept to see what links it might have to propagate. It makes a replica of itself (by calling the copy constructor) and places it in the destination concept. The destination concept is retrieved by a call to the Concept Manager which will return a reference to it. The time to get the reference depends on whether the looked-up concept is already in memory or not. The new destination is stored in the member variable visited that keeps track visited nodes (actually its path so far). Then, the current replica is sent to the new destination using the same acceptMarker message.

4. Go Further. At this point it sets the destination of the remaining link as the new host (it makes replicas for all links but one).

5. Filter paths by reporting only the most specific concept in the hierarchy at which the markers met for the first time

6. Filter paths by using Method 1

7. Report Paths

8. Print paths and inferences

The filter process (steps 5 and 6) is actually embedded in the propagation rule. When a collision is detected it will be reported only at the most specific concept (this is done by checking that there is not more than one common host visited along the ISA links). Than we verify if the current path explains the lexical relation from the original text (stored in the where-to-go field).

The novelty of this execution mechanism is that the concepts may be regarded as environments on which processing elements (markers) are allowed to process the local information. The execution flow is embedded in the propagation rules. The scheduling of the threads is done by controller, namely Java Run Time Environment.
Table 4.5. Memory Usage without heuristic

<table>
<thead>
<tr>
<th>Propagation wave</th>
<th>Number of Threads</th>
<th>Number of active objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>211</td>
<td>69</td>
</tr>
<tr>
<td>2</td>
<td>1349</td>
<td>144</td>
</tr>
</tbody>
</table>

4.4. Experiments

We allow the markers to propagate along all links starting from a concept. The marker spawning is exponential and will soon dominate the memory resources. We give in Table 4.5 dynamic statistics of this experiment.

As shown in this table the number of threads is exponentially increasing as the propagation progresses, while the activation of the concepts in memory is increasing almost linearly. The markers will soon dominate the system and this will lead to an out of memory message. The explanation stands in the branching factor associated with one marker. For example move\#1-5(v) has 133 links. We must restrict the spawning factor of the markers at one node by applying some heuristics. We applied the following restrictions:

- Propagate only along ISA and GLOSS relations. That means up in the hierarchy and through the gloss relations. The number of isa relations is 71,940 (from [19]), the most numerous, after the synset relations.

- Look forward when making a replica look forward to the destination to check for a cycle. So far we checked for cycles only at arrival.

We have obtained the results in Table 4.6.

In this table the values represent cumulative values of the threads and, respectively, of the active base. By applying this heuristic we gained a good balance between the memory usage for knowledge base and the memory usage for markers. The num-
Table 4.6. Memory usage with the proposed heuristic

<table>
<thead>
<tr>
<th>Propagation wave</th>
<th>Number of Marker Threads</th>
<th>Number of active objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>89</td>
<td>69</td>
</tr>
<tr>
<td>2</td>
<td>242</td>
<td>93</td>
</tr>
<tr>
<td>3</td>
<td>349</td>
<td>131</td>
</tr>
<tr>
<td>4</td>
<td>412</td>
<td>166</td>
</tr>
<tr>
<td>5</td>
<td>473</td>
<td>179</td>
</tr>
<tr>
<td>6</td>
<td>496</td>
<td>189</td>
</tr>
</tbody>
</table>

The number of markers in memory are always increasing, because we are never killing them but only stop them. This is because we want to retrieve their paths in the case of a collision. For a spawning factor bigger than 3, assuming a random choice of the link types, we get the same out of memory message. The number of active threads is twice as big as the number of active marker threads (those operating at one level) because an active marker has a monitor associated with, and this is implemented as a thread, whose role is to detect when the marker’s propagation rule is stopped and to send the marker further. When there are no more links to go or the marker reaches a top concept and it has no more links to propagate along, it is sent to a fictive host (the marker manager). When a collision occurs, the other marker (the other half of the joined path) will be retrieved via the host concept and the Concept Manager.

By applying the proposed heuristic, we were able to balance the marker spawning and the amount of knowledge base in memory. However the restrictions imposed were too strong. We are looking forward to allowing markers propagate along other types of links. Because of importants requirements in terms of memory and computational resources, a distributed implementation could be regarded as a possible solution to feed in these requirements either using the distributed facilities of Java or in a distributed Java virtual machine.
CHAPTER 5
CONCLUSIONS

We presented in this thesis a framework for developing marker-propagation based solutions to different types of problems. We defined a parameter $k$, the marker decaying factor, and we analysed the behaviour of two applications: pattern matching and text inference. Using the parameter $k$ we may classify applications into:

- Strong constrained applications ($k>1$); these are applications with strong constraints embedded in the propagation rules. Because of the constraints the markers are dying rapidly. The pattern matching presented in chapter 3 belongs to this class. The larger the semantic network in memory the more solutions are found. If the semantic network in memory is too small, we may not have gotten any solutions because of the missing nodes from memory. The larger the semantic network in memory the more solutions would be found. For this class of applications, with $k$ larger than one, we proposed a heuristic that increased the size of the semantic network in memory up to 100%, by taking advantage of the memory freed up by the dying markers. For the pattern matching application, we obtained a 33% gain in the network size loaded in memory.

- Weak constrained applications ($k<1$), like text inference, in which the marker spawning is significant and tends to dominate the memory resources, leaving a small portion for the semantic network. By restricting the links over which a marker may spawn, and applying a lazy loading of the knowledge base into memory we managed to balance the memory resources allocated for markers, respectively nodes of the semantic network. At that point we were able to get results, in the form of paths among concepts. Most of the paths are not
relevant. Using filtering methods like those developed in [19] we filtered out the paths and obtained inferences (see Table 5.2). Method 1 retained 3 paths out of 25 for the first sentence of the input text. By using the marker propagation mechanism, the speedup gained over a sequential approach is significant: 49.58. There are 6.67 concepts per path with an average of 3.35 steps for finding a path. Because all the paths are processed in parallel this is the total time for finding all the paths using the marker parser. In the serial case we would have had to reach 166 concepts with one step per concept. Dividing 166 to 3.35 we get 49.58, which is the mentioned speedup. In this computation, we did not consider common concepts to different paths and synchronization aspects.

We look forward to improving the behavior of the last class of applications by applying better heuristics and to use more resources available in distributed environments. In the case of text inference application we spent a lot of time for building the knowledge base. First of all we built the WordNet server to access all the data in WordNet from within Java programs. Then we disambiguated the input text and
the glosses using the software and methods developed in [34][28]. After that we checked the lexical relations inside glosses as indicated in [18]. We are expecting also automated systems to deal with those issues of building the knowledge base. The framework presented in Chapter 1 proved to be useful for two important classes of applications. The memory requirements are its main drawback. An implementation of the framework in a network of workstations would be one future extension of this thesis. The parallelism available in our implementation of the marker propagation model can be fully exploited using a multiprocessor architecture which will allow different threads be scheduled on different processors. In a uniprocessor machine, threads will share the CPU, running one at a time according to a scheduler, and there will be no significant speedup.
REFERENCES


