A project Report
On

Google N-gram Patterns

For CS 8621

Advanced Computer Architecture

Fall 2007

Team Name: Flamengo

Team Members:

1. Darshan Paranjape.
2. Bin Lan
3. Vishnu Pedireddi
4. Anurag Jain

Department of Computer Science

University of Minnesota, Duluth

TABLE OF CONTENTS

ABSTRACT

This project presents an easy and fast way to analyze Google n-gram data, which is contributed by Google Inc. Google n-gram data consists of a huge amount of word information based on real life searching queries entered by internet users. The huge amount of data makes it so hard to analyze the whole data set. In this project, we present a possible parallel solution to build and access co-occurrence network using Google n-gram data. Moreover, we use the co-occurrence network to find relationship (path) between words in this large corpus. We also build a common library based on C/MPI for all the similar co-occurrence network analysis programs. This method was tested on both Blade system and Altix system from MSI at University of Minnesota Twin City campus.

1. INTRODUCTION

Google n-gram data consists of 1,024,908,267,229 tokens, 13,588,391 unigrams, 314,843,401 bi-grams, 977,069,902 trigrams, 1,313,818,354 four-grams and 1,176,470,663 five-grams. It is being stored on 6 DVD set. The total size of the n-gram data is about 27.6 GB. The huge quantity of the data set makes it very hard to access or retrieve any kind of useful information. A modern computer usually contains a memory system of size 2 to 5 GB, which means it is impossible to store the whole Google n-gram data into the main memory. An alternative would be to use a lot of file I/O to access the data directly from the files where it is being stored. However, the speed of file I/O is significantly slower than access the main memory. So this approach would require a lot of waiting time and it is very inefficient.

Our approach is to utilize the modern parallel computer so the Google n-gram data could be distributed to different machines. Hence, it is possible to store the whole Google n-gram data (co-occurrence network) into the main memory in different machines. This method provides a possible solution to access the data with a very small time requirement. Because the network is sitting inside the main memory, we can discover relationship (path) for a target word on the fly. Fig. 1.1 demonstrates the components of our system.

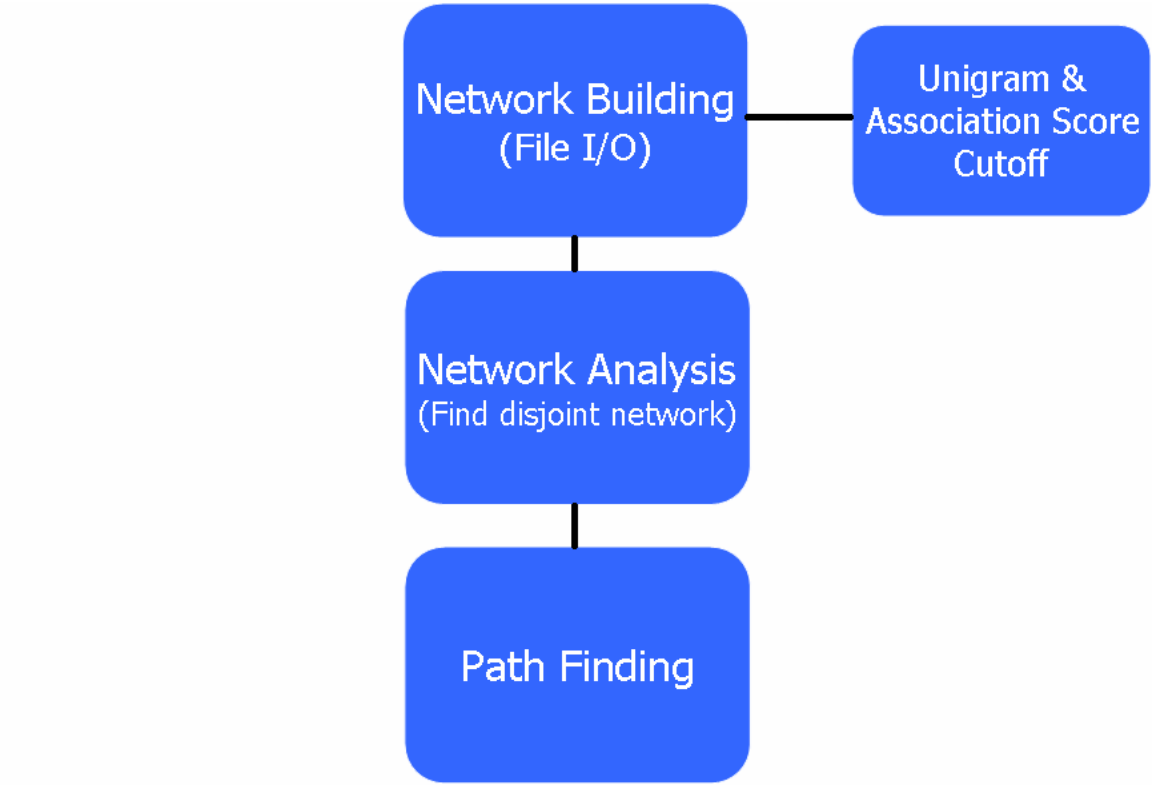


Fig. 1.1

First, the system reads the Google n-gram data from files to build the co-occurrence network. The network information is stored in the main memory within selected data structures. Note that the Unigram cutoff and association score cutoff is implemented while building the network. After reading files and building the network, the system gathers the network information and finds the disjoint networks. Finally, the system reads target words from user query file and finds the paths of specific path length by accessing the network.

2. CO-OCCURRENCE NETWORK

A co-occurrence network is a network that links together words that have occurred together in sentence. Each word represents a node in the network, and the edges between the words indicate that they have occurred together. The weights on the edges are the frequency of the two words occurring together in that order. Edges connecting the words together are directed edges. So each node can have two types of edges associated with it namely, outgoing and incoming. Outgoing edge of one node has to incoming edge of some other node.

For example consider following sentence

"It is very important"

In co-occurrence network built over above sentence, 'It' should be connected via a directed edge that points to 'is', 'is' should be connected to 'very', and 'very' should be connected to 'important'. Once a co-occurrence network is built with required node and edge information, this network can be traversed through directed edges. If we consider a word i, all the words that can be reached from that word through network traversal are said to be in same co-occurrence network as that of word i. Thus in a given dataset there can be more than 1 co-occurrence network. Words in different co-occurrence networks have no association between them & we cannot traverse from one word to other in this case.

Following diagrams show a sample co-occurrence network built on 2 sentences

"It is very important"

"It is bad"

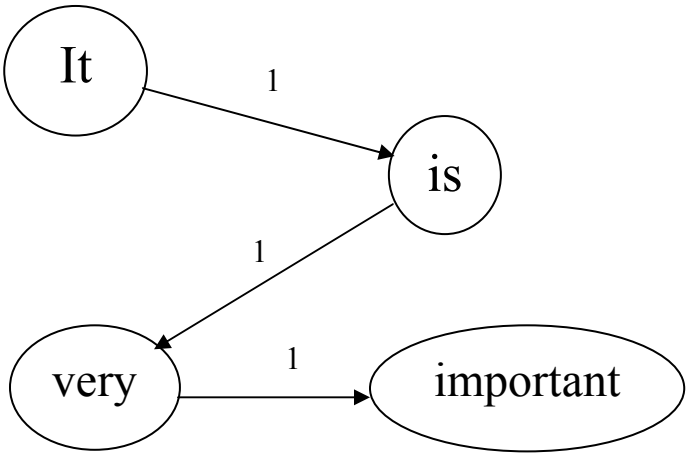


Fig. 2.1
Co-occurrence network built on the sentence "It is very important".

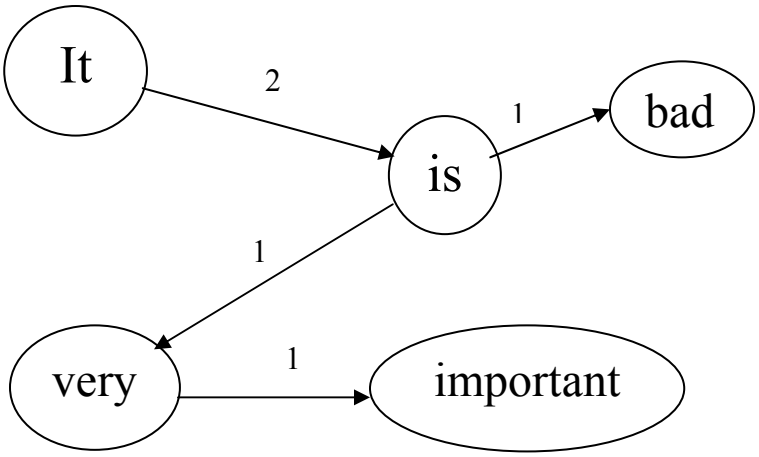


Fig. 2.2
Co-occurrence network built on the sentences "It is very important" and "It is bad".

In above figures, the number associated with edges is the frequency of words occurring together. In fig.2.1, nodes are created for all the words in first sentence. Word pair "It is" occurs once in first sentences, so edge frequency is shown as 1. Fig 2.2 shows the updated co-occurrence network after reading second sentence. When word pair "It is" occurs again in the second sentence, its edge frequency is incremented to 2. As nodes for "It" and "is" already occur in the network, single node for "bad" is added to the network.

3. RELATED PREVIOUS WORK

3.1 Co-Occurrence vectors from corpora vs. distance vectors from dictionaries [1]

This paper presents comparison between co-occurrence vector & vectors derived by measuring the inter word distances in dictionary definitions. A word vector is defined as the list of distances from a word to its origins. Origins of a word are the words that occur in its definition. For each word in the word vector, a distance vector is formed where i-th element is the distance (length of shortest path) between word & its i-th origin. Length of link between 2 words depends on total number of links to & from each word and number of direct links between these 2 words. If links to & from a word are less then it's a low frequency or rare word. A path between low frequency words indicates strong relation and thus considered as short path.

A co-occurrence vector of a word is defined as the list of co-occurrence likelihood of the word with a set of origin words. Co-occurrence likelihood of a word depends upon occurrence density of the word in whole corpus and density of that word in neighborhood of an origin word. The co-occurrence vectors are more useful than distance vectors while considering word sense disambiguation based on context similarity. But the property of positive-or-negative is reflected more strongly in distance vectors than in co-occurrence vectors.

3.2 Conceptual Grouping in Word Co-Occurrence Networks [2]

This paper presents a technique called conceptual grouping that can be used to distinguish between different meanings of user queries given on a document collection. A co-occurrence network is built on text database from given collection. This network is used to find groups of words semantically related with the user query. These groups are used to reorganize the results according to what the user has meant by his query. For Example, if a user enters query 'water', all words that are linked to it in the co-occurrence

network are likely to be semantically related to water. It is likely that there are more documents in which 'water' appears close to these words. So this method may help users to find documents more easily.

The co-occurrence network is built on all words in the textual database. Words i and j are said to be in co-occurrence in document d only if they both appear in d, no further than 50 words apart. Associative strength between words i and j are found using bounded add operator. Then relevance between 0 & 1 are found for i and j against all documents in the collection. Use of co-occurrence in this approach helps to retrieve information by an analysis of what the user meant in his query.

3.3 Using co-occurrence network structure to extract synonymous gene and protein names from MEDLINE abstracts [3]

This paper discusses a new text-mining method based on analyzing the network structure created by symbol co-occurrences as a way to extend the capabilities of knowledge extraction. Text mining and knowledge extraction are ways to aid biomedical researchers in identifying important connections within information in the large biomedical knowledge base. The method was applied to the task of automatic gene and protein name synonym extraction.

In the first step of co-occurrence network, nodes are gene and protein names and symbols, and edges are labeled with the number of times the connected names have occurred in a text source together. Occurrences of gene and protein names are then replaced with a regular expression that matches a wide variety of possible gene and protein names to construct a gene name synonym network. Synonym co-occurrence networks will have many separate, internally tightly linked clusters, since synonyms of synonyms should also have co-occurrences in the network. Once the co-occurrence network is built, synonym pairs can be extracted from the network using a graph traversal algorithm like Dijkstra's shortest path algorithm.

3.4 Choosing the Word Most Typical in Contest Using a Lexical Co-occurrence Network [4]

This paper presents a very useful application of co-occurrence network constructed by word tokens. The research group uses information from the part-of-speech-tagged 1989 Wall Street Journal as the corpus to build the co-occurrence network. The nodes in the network are the words from corpus. They used mutual information scores and t-scores to determine the relations between words in the networks. Namely, they were using these two functions to build the first-order edge information in the network. Meanwhile, they also use the sum of t-scores of the shortest path between two words to determine the second-order relation between these two words.

By building such co-occurrence network, the group believes they could present the potential subtle difference between synonyms. Giving a sentence and a gap in the sentence, one could decide the most "typical" word from a set of synonyms candidates by looking at the score of co-occurrence relations between the candidates and the words of the sentence in the network. However, as indicated by the author, it is very hard to decide which candidate is the "right" answer since there is not absolute answer to this question.

3.5 Discovering word senses from a network of lexical co-occurrences [5]

The main purpose of the paper is to introduce a new way of making word senses from a network of lexical co-occurrences tested on French and English. This method tire to eliminate the short falls and criticism faced by lexico-sematic networks such as WordNet for the way they define word senses.

The main drawbacks in the conventional networks occur in the way they form relations between synonyms, hyponyms and hyperonyms. The solution provided here mainly focuses on building a distribution list based on classes of equivalent words and not limiting just to the synonyms, hyponyms and hyperonyms. For each word the method runs a clustering algorithm to find all its instances. This then forms the sense of the target word by gathering the co-occurrents based on the similarity.

4. GOOGLE Inc. N-GRAM DATA

An N-gram is a sub-sequence of n items from a given sequence. The items can be letters, words or base pairs according to the application. An N-gram of size 1 is a "Unigram"; size 2 is a "Bi-gram"; size 3 is a "Trigram"; and size 4 or more is simply called an "N-gram". N-grams are used in several areas of computer science, computational linguistics, and applied mathematics.

Google Research has been using word N-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction etc. Google released "version 1" of their N-gram word data sets through the Linguistic Data Consortium. This data set contains English word n-grams and their observed frequency counts. The length of the Google n-grams ranges from unigrams to five-grams. The n-gram counts were generated from approximately 1 trillion word tokens of text from publicly accessible Web pages. The input encoding of documents was automatically detected, and all text was converted to UTF8.

For $150 US, one can get the Google n-grams of following size on a 6 DVD set.

File sizes: approx. 24 GB compressed (gzip'ed) text files

```
Number of tokens:    1,024,908,267,229
Number of sentences:    95,119,665,584
Number of unigrams:         13,588,391
Number of bi-grams:         314,843,401
Number of trigrams:        977,069,902
Number of four-grams:     1,313,818,354
Number of five-grams:     1,176,470,663
```

The following are the examples of the N-gram data present in this corpus:

Unigram

```
Carfan7     389
```

Bi-gram

```
Carfind.ca Carfest      309
```

Tri-gram

```
ceramics collectables collectibles    55
```

Four-gram

```
serve as the incoming  92
```

Five-gram

```
It is very important for       63
```

5. ALGORITHMS

5.1 Data Structures and Building the Network

5.1.1 Data Structures for Network Creation

Co-Occurrence network created in this project depends mainly on 2 data structures. Data structure "node" stores the information about the words in the Unigram data. This information includes word, index of the word in unigram array, total number of outgoing edges, list of all outgoing edges and list of all incoming edges. Following is the "node" structure.

```
typedef struct node
{
	char* token;		//Word
	edge *incoming;		//Starting pointer of incoming linked list
	edge *curr_incoming;	//Current pointer of incoming linked list
	edge *outgoing;		//Starting pointer of outgoing linked list
	edge *curr_outgoing;	//Current pointer of outgoing linked list
	int has_seen;		//Variable used while finding distinct networks
	int is_checked;		//Variable used while finding distinct networks
	int index;		//location in unigram array
	int count_outgoing;	//Total number of outgoing edges
	int count_incoming;	//Total number of incoming edges
	long int total_out_weight; //Sum of all outgoing weights
	int incoming_nodes_added;  //Variable used in the beta stage
	int outgoing_nodes_added;  //Variable used in the beta stage
};
```

A word can have more than 1 incoming or outgoing edge. So we need to store the information of all incoming and outgoing edges for a word. We have created a structure "edge" that stores this information. The easiest option is to create an array of edge structure. But before reading the Bi-gram file, we don't know the total number of

incoming and outgoing edges for the word. So we are maintaining a linked list for both incoming and outgoing edges. Following is the "edge" structure.

```
typedef struct edge
{
       int index;          //location in unigram array
       long int freq;      //weight associated with edge
       struct edge *next;  //Pointer to next entry in the linked list
};
```

5.1.2 Building the Co-occurrence Network

Main task while building the Co-Occurrence network is to read the Google N-gram data and to store the relevant information in the data structures mentioned in above section. The size of the Google N-gram data set is very large. So creating the Co-Occurrence network in real-time is a difficult challenge. This challenge can be handled by use of parallelism.

If only a single processor reads all files, allocates memory, stores the network information, if will have issues related to both time & memory. Considering the size of the data, it is not possible for single processor to allocate memory for storing network information.

We can think of network creation in terms of 3 tasks. First, to perform file I/O; second, to process the data from file I/O; third, to store the processed information in related data structure. Even though size of the unigram & bi-gram files is very large, file reading alone does not require very long time. Also storing the processed information in the data structure doesn't take much time. But the most important task of data processing requires memory as well as time in large amounts. Hence this task has to be done in parallel to serve both memory & time requirements.

In the approach we are using for network creation, we let each processor read only a part of Bi-gram file. Each processor performs 3 tasks of network creation. But data involved in each of these tasks reduces considerably.

While creating the Co-occurrence network for Bi-gram data, we are making use of both Unigram and Bi-gram data files. The main idea behind this is to maintain array of all unigram tokens which will be used as reference while reading Bi-gram data.

Following are the steps involved in building co-occurrence network.

Step 1: Count the number of unigrams.

We're not hard coding the number of unigram so we can work on test data set. In this step, the whole unigram file is read to count the number of lines. The number of lines in this file gives us the total number of unigrams. It is necessary to find this size first so we can allocate memory to array of node structures.

Step 2: Memory Allocation.

In this step, we allocate array of node structure with size equal to number of unigrams. Also the structure variables are initialized to proper values during this step.

Step 3: Read the unigram file.

In this step, unigram file is read again to store information to the node structure. Every unigram is stored in the 'token' field of the node structure. Also 'index' field stores the index of the unigram in array of node structures.

Step 4: Bi-gram File Distribution.

The size & number of Bi-gram files is very high. Also while reading the file; we need to store the edge information which is going to take considerable amount of time. So we decided to implement parallelism in this step. The main idea is to assign each processor some number of lines in a file instead of making each processor read the whole file. There can be 2 approaches while distributing lines to the processors. We can allocate

blocks of size (number of lines/number of processors) or use interleaved distribution where for p number of processors processor k gets line p,p+k,p+2k. We've used block allocation method as it faster option. After this calculation, every processor gets the starting line & ending line for each Bi-gram file. Every processor skips the lines in file until it reaches starting line. It reads the files & processes data only until it reaches ending line.

Step 5: Finding index from Unigram array

After reading both the Bi-gram words on a line, we need to find their indices in the Unigram Array. With this indices associated with word, it can be search faster in network analysis. Finding index from array of large size (13 million approx) with liner search is method is very time consuming. We are using binary search algorithm with time complexity O (log (n)). A binary search algorithm is a technique for finding a particular value in a sorted list. We can make use of this algorithm as token in the node structure array are in alphabetical order.

Step 6: Adding incoming and outgoing edge information

After reading 2 words from a line and finding their index, we need to store incoming and outgoing edge information. We store the index of the second word & weight of the edge in the outgoing edge linked list of first word. Similarly index of the first word & weight of the edge is stored in incoming linked list of second word. Every processor does this for its allocated number of lines.

The co-occurrence network is built after above 6 steps. As this is implemented in parallel, all the incoming & outgoing edge information of a single node might not be available in single processor. So we can say the Co-Occurrence network is distributed in different processors. Distributing the network in different processors makes network creation a very fast step. But after the building the network in this manner, there has to be communication between processors to access a particular node.

5.2 Analyze the Network and Find Disjoint Networks

To analyze the network, we use a two steps algorithm. First step is a local analysis, which finds the disjoint networks based on the local edges information. The second step is a parallel reduce function which gather the local network information from all processors and then derived the global result.

5.2.1 Local Analysis

In our data structure, all nodes are stored in an array in every processor while the edges are totally distributed among processors. Each node in the network has two checking bits, one is called has_seen, and another is called is_checked. The algorithm starts checking with the first element of the nodes' array. If one node's has_seen bit is not checked, then it will mark the node as has_seen. For all the nodes that directly connect to current node, the algorithm then marks them as has_seen and repeats the same steps for each of them. After checking all the direct-connected nodes, the algorithm marks the current node as is_checked. The algorithm is terminated if all the nodes in the array are marked as is_checked. After the process, each node's is_checked bit is also the network identify number. Notice that all disconnected nodes will have network ID 0. Since each node has the information on both incoming edges and outgoing edges, one call of the algorithm can guarantee to travel through every nodes in the connected network and mark all the nodes with the same network ID. Hence, we can gather the information of the network.

```
count :-> 0;
1. for all nodes in the network
2.      if node m has not been seen before
3.              Mark m as has_seen;
4.              for all the nodes that directly connect to m
5.                      Mark all the nodes with has_seen;
6.                      Repeat step 3 with all the nodes;
7.              End for
8.              Mark node m as is_checked;
9.      End if
10.     count:->count + 1;
11. End for
```

Fig. 5.2.1
Algorithm for local analysis

5.2.2 Global Analysis

After the local analysis, every node's is_checked bit actually stores the network ID. However, the edges information is distributed among all processors in our system. Each processor has a unique but incomplete view of the network based on the edges information it has. In example 5.2.2.1, after the local analysis CPU 1 has the information about node A is connected to node B, C, and D in local network 5. For CPU 2, however, node A might be connected to node B, E, and F in local network 2. Note that for different processors, the same network ID does not mean they are connected. For example, node A in CPU 1 might have network ID 1 and node B in CPU 2 might also have network ID 1. But it does not mean node A and node B are connected in the global network. It is opposite to the nodes' index. For all processor, the same node always has exactly same index in the nodes' array.

Example 5.2.2:

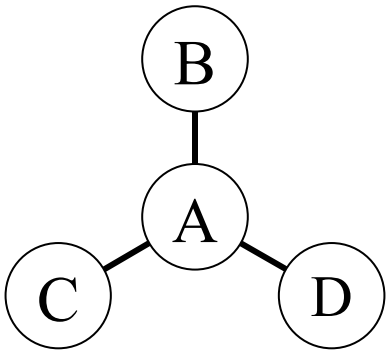


Fig. 5.2.3
In CPU 1, A, B, C, and D are connected with local network ID 5.

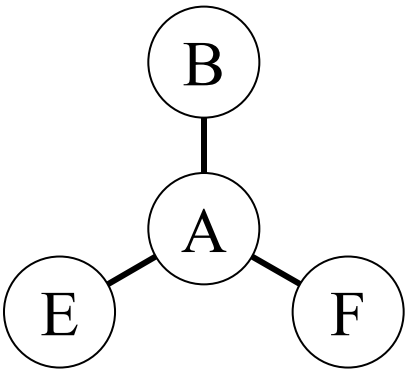


Fig. 5.2.4
In CPU 2, A, B, E, and F are connected with local network ID 2.

The main goal of our global analysis is to bring the local network information together so we can form a layout for the whole network. The algorithm is actually very straight forward. Once we see a node connecting to different sets of nodes in different processors, we mark these sets as one network because they all connect to a common node. In

previous example 5.2.2, once the program notices that node A connects to two different sets of nodes in CPU 1 and CPU 2 it will assign a new global network ID to these two sets of nodes.

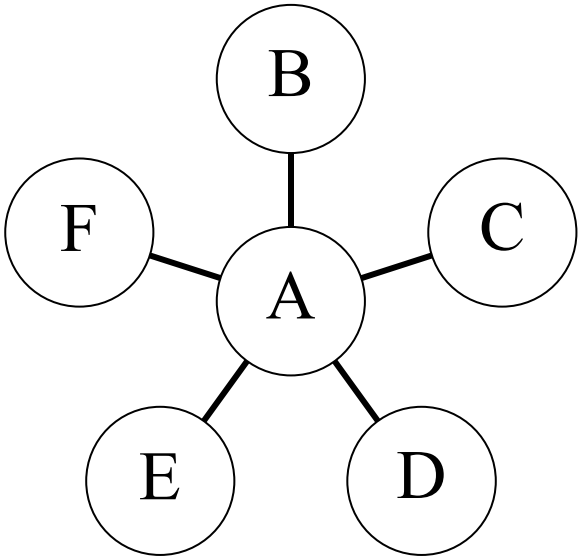


Fig. 5.2.5
After global analysis, two networks are combined and assign a new global network ID.

5.3 Find paths of specific length for target words

Once we have constructed the co-occurrence network it is time to query the network.

Beta Stage Requirements Fulfilled

Allow a user to specify a target word, and display the paths of a given length leading to and from that word (and to the words that are connected to those words, and so on).
Thus, the specified target word should be at the center of the paths that lead into and out from it. Path lengths are defined in terms of the number of edges in the path to and from the target word. Thus, if a path length of 2 is requested, that should cause you to display all the paths going into (and out of) the target word that have a total length of four edges (where the target word is in the middle of two paths of length 2). Please consider the possibility that a word requested may not appear in the network, in that case you should indicate that the word was not found. Your search should be case sensitive, that is "to" and "To" should be treated as different words. To display each path to and from the target that is path-length edges long in a format similar to the following:

word-2->(weightx)->word-1->(weighty)->target->(weighta)->word1->(weightb)->word2

Note that path-length in this case is 2. weightx is the frequency of "word-2 word1", weighty is the frequency of "word-1 target", weighta is the frequency of "target word1", and weightb is the frequency of "word1 word2". TIME is the amount of time (in seconds) it took to find the path.

Don’t print cyclic paths. (Cyclic paths occur when same edge occurs in a path more than once.)
Print complete paths. (If the last node in the printed path does not have any parent or a child or both then that path is called a complete path.)

Approach

To print the specified path length we need to have the disjoint network for that particular token as well as other tokens connected to this token and so on. The disjoint network is presently distributed among the processors used to build the network. To be able to print a result we have to make sure that the required disjoint network should be present on one processor. So, we have to collect the edge information on some processor and then print the paths. We chose processor zero to be the processor that will store all the edge information and print the final results. We call this processor to be the master processor.

Algorithm

We will try to understand the algorithm using an example:

Suppose, the files to read are:

1gm File

A 1000
B 2000
F 3000

2gm file to read

A A 100
A B 200
B A 400
B B 500
F A 600

Network Builder Reads The 2gm-File In Parallel (Number Of Processors: 2)

We are reading only one file here but the same operations will be performed if there are more files.

Once, the 2gm file is read by the processors in parallel the structure formed in the memory is shown in FIGURE (1) 1.

Processor 0	Processors 1
|A| -> 0,100->1,200 (Outgoing)	|A| -> NULL (Outgoing)
-> 0,100->1,400 (Incoming)	-> 2,600 (Incoming)
|B| -> 0,400	|B| -> 1,500
-> 0,200	-> 1,500
|F| -> NULL	|F| -> 0,600
-> NULL	-> NULL

FIGURE (1)

Here, |x| represents a token in the unigram array where x could be A, B, F. Each processor has a unigram array with it. The token A has location 0, token B has location 1, and F has location 2 in the unigram array (represented in the form of the column).

Now, to represent the edges spreading out of a token we have kept two linked lists one represents outgoing edges and the other incoming edges.

In the figure above, we first have the outgoing linked list with the token and then the incoming linked list.

To represent a bigram in the 2gm file. We will have two entries in the linked lists:
One outgoing
One incoming

For Eg:- To represent the edge A B 200
We have an entry in the outgoing linked list for token A in the unigram array for processor 1. (|A| -> 1,200 (Outgoing Linked List))
Also, we have an entry in the incoming linked list for token B in the unigram array for processor 1. (|B| -> 0,200 (Incoming Linked List))

Note:-

1) We have specified processor 1 because this line in the 2gm file had been read by this processor and it will be having its information and no other processor.

2) One more thing to notice is we are keeping the indices of the tokens from the unigram array in the linked lists instead of the token itself as it would be requiring a lot of memory.

Path Finding

Now, we have the network built with us.

Suppose, we have to print paths of length 2 with 'A' at center.

As can be seen in FIGURE (1) the disjoint network for 'A' is distributed among the two processors. So, we collect the network somewhere say processor 0 (master processor).

The important point to note here is:-

We are not required to collect the whole disjoint network to which 'A' is connected; we only need to collect for the specified length.

So we collect all the incoming and outgoing edges of token 'A' as well as B and F on processor 0. And FIGURE (2) displays the result of that.

Processor 0	Processors 1
|A| -> 0,100->1,200 (Outgoing)	|A| -> NULL (Outgoing)
-> 0,100->1,400->2,600 (Incoming)	-> 2,600 (Incoming)
|B| -> 0,400->1,500	|B| -> 1,500
-> 0,200->1,500	-> 1,500
|F| -> 0,600	|F| -> 0,600
-> NULL	-> NULL

FIGURE (2)

How to avoid self-loops and print complete paths?

Question: What are self-loops?
Answer: Self-loops occur when same edge occurs in a path more than once.

Avoiding Self-loops (Edge-Marking Method)

In this method while building up a path to be printed we mark all the edges that have been included in the path so that if they occur again we can just skip them and move to the next connected edge and so on till we find an unmarked edge. In this was way we can avoid the self loops.

Question: What are complete paths?
Answer: If the last node in the printed path does not have any parent or a child or both then that path is called a complete path.

Note: We will be printing such paths. It may happen that complete paths do not satisfy the specified length criteria i.e. that is their length is smaller than the specified length but we will still print them.

Printing Complete Paths

To print the complete paths we took advantage of the property of complete paths that the last node in a complete path will have no parent or no child or both. If the last node does not have any child or parent or both and the length is less than or equal to the specified length we printed the path.

Now we are ready to print some paths:

Figures below represent the printing procedure.

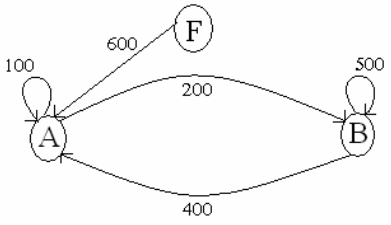
Print Path For ‘A’ And Length 2

NETWORK PROCESSOR 0 FORM

Processor 1

|A| -> 0,100->1,200 (Outgoing)
-> 0,100->1,400->2,600(Incoming)
|B| -> 0,400->1,500
-> 0,200->1,500
|F| -> 0,600
-> NULL

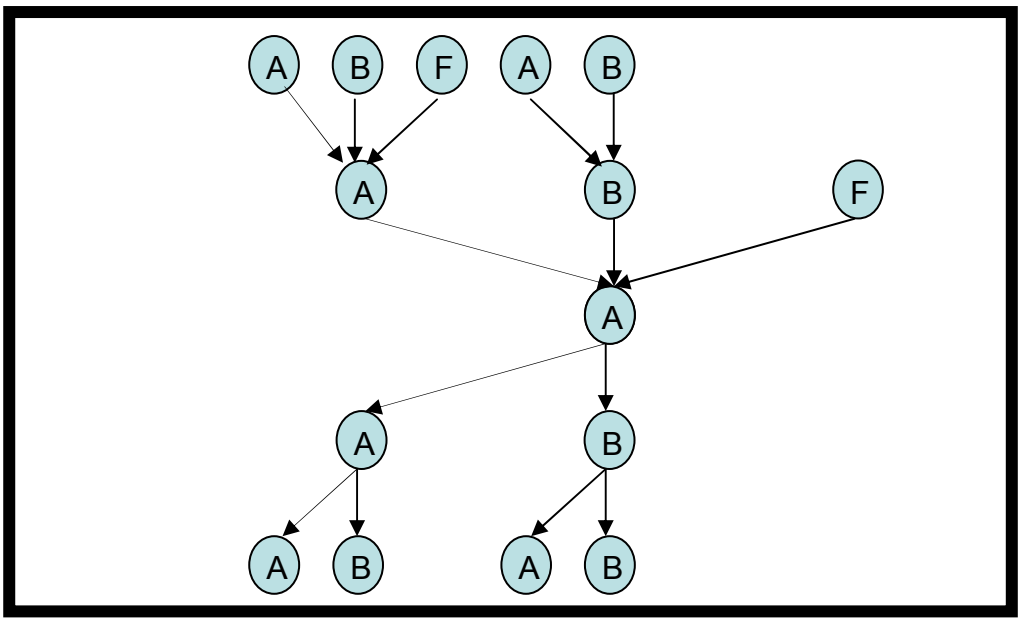
DISJONT NETWORK GRAPH FORM



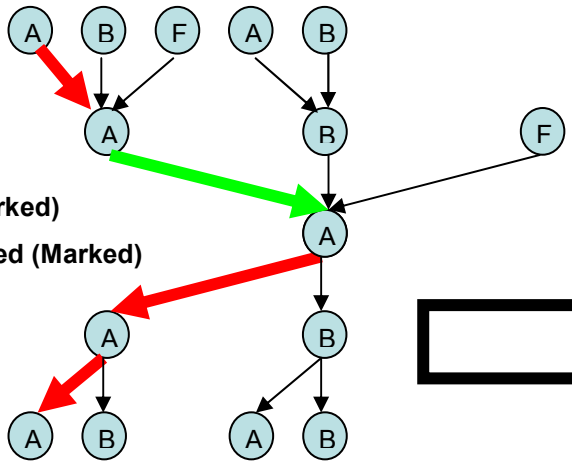
(FIGURE 1)

Print Path For ‘A’ And Length 2

A’s LENGTH 2 NETWORK



Print Path For ‘A’ And Length 2



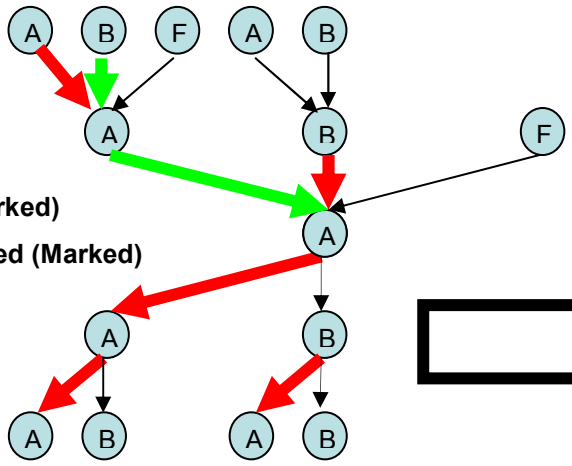
Query
Target: A
Length: 2

- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

A->(100)->A->(100)->A(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



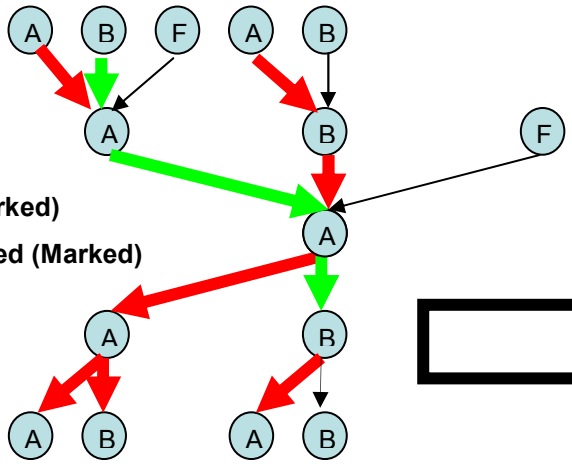
Query
Target: A
Length: 2

- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(100)->A(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



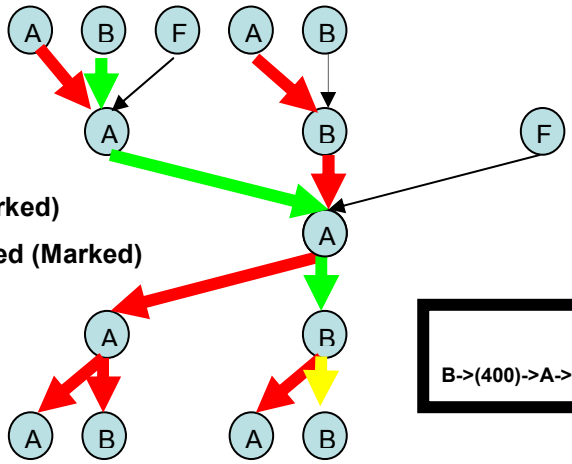
Query
Target: A
Length: 2

- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(100)->B->(400)->A(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



Query
Target: A
Length: 2

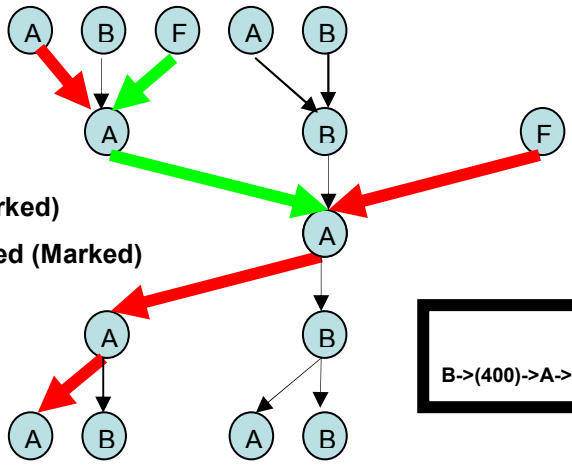
- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B

B->(400)->A->(100)->A->(100)->B->(500)->B(PRINTED)
(CENTER)

Print Path For ‘A’ And Length 2



Query
Target: A
Length: 2

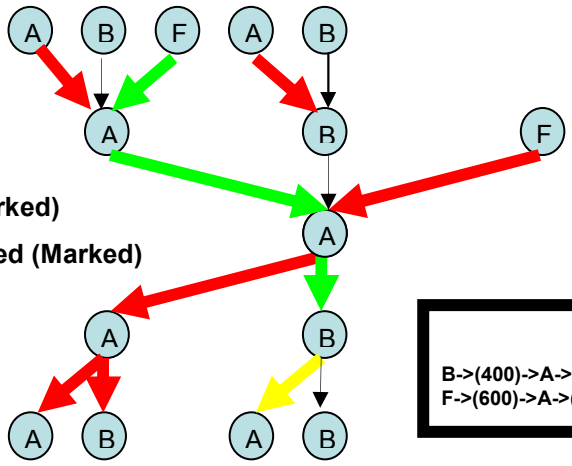
- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B

F->(600)->A->(100)->A->(100)->A(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



Query
Target: A
Length: 2

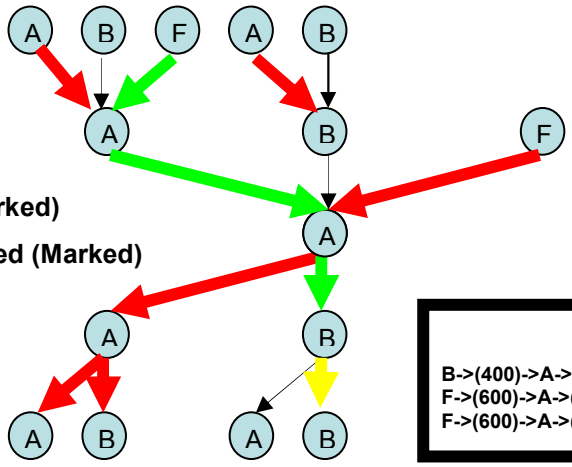
- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A

F->(600)->A->(100)->A->(200)->B->(400)->A(PRINTED)
(CENTER)

Print Path For ‘A’ And Length 2



Query
Target: A
Length: 2

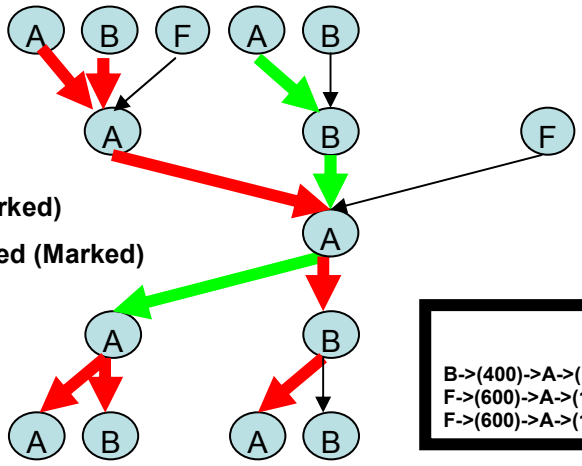
- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B

F->(600)->A->(100)->A->(200)->B->(500)->B(PRINTED)
(CENTER)

Print Path For ‘A’ And Length 2



Query
Target: A
Length: 2

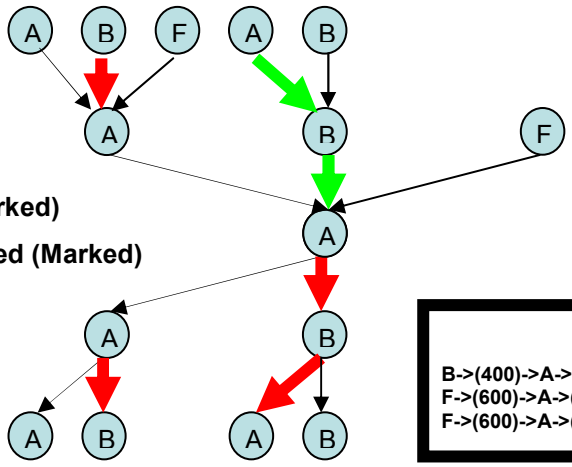
- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B

A->(200)->B->(400)->A->(100)->A->(100)->A(SKIPPED)
A->(200)->B->(400)->A->(100)->A->(200)->B(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

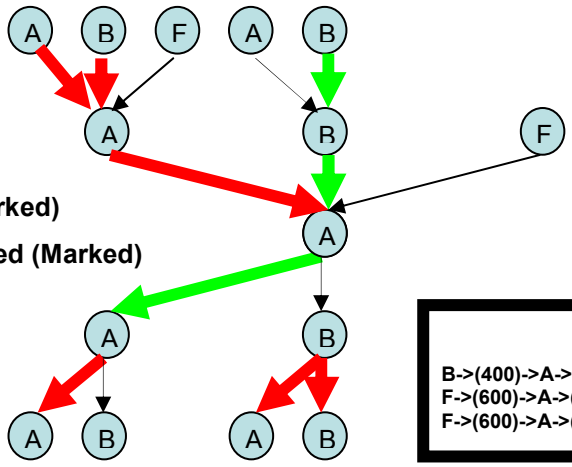
Query
Target: A
Length: 2

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B

A->(200)->B->(400)->A->(200)->B(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



Query
Target: A
Length: 2

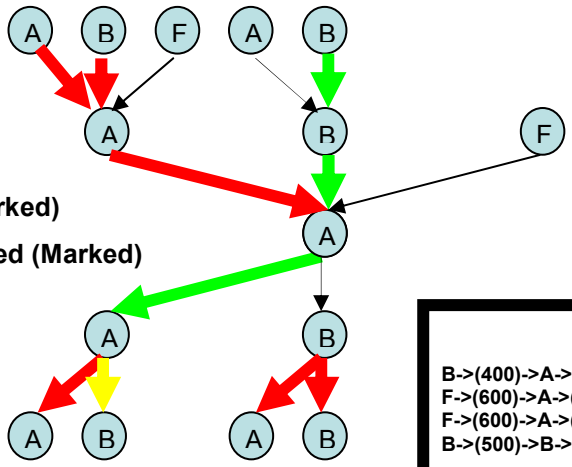
- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B

B->(500)->B->(400)->A->(100)->A->(100)->A(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



Query
Target: A
Length: 2

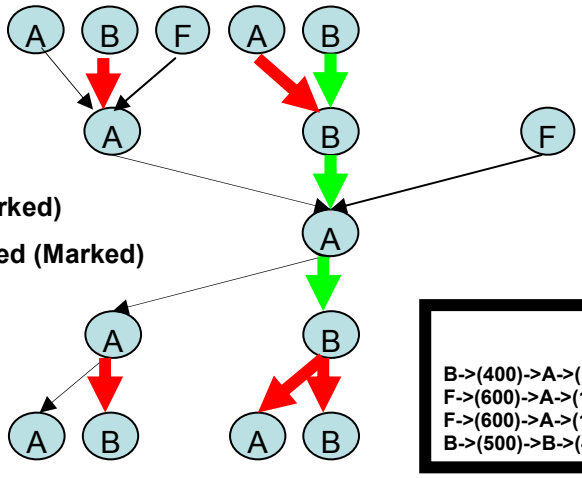
- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B
B->(500)->B->(400)->A->(100)->A->(200)->B

B->(500)->B->(400)->A->(100)->A->(200)->B(PRINTED)
(CENTER)

Print Path For ‘A’ And Length 2



Query
Target: A
Length: 2

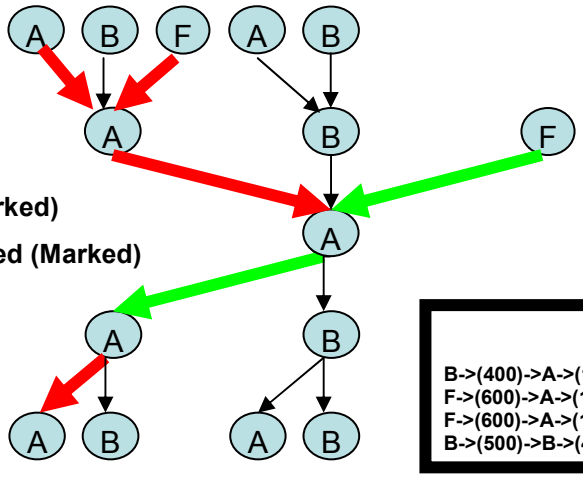
- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B
B->(500)->B->(400)->A->(100)->A->(200)->B

B->(500)->B->(400)->A->(200)->B->(400)->A(SKIPPED)
B->(500)->B->(400)->A->(200)->B->(400)->B(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

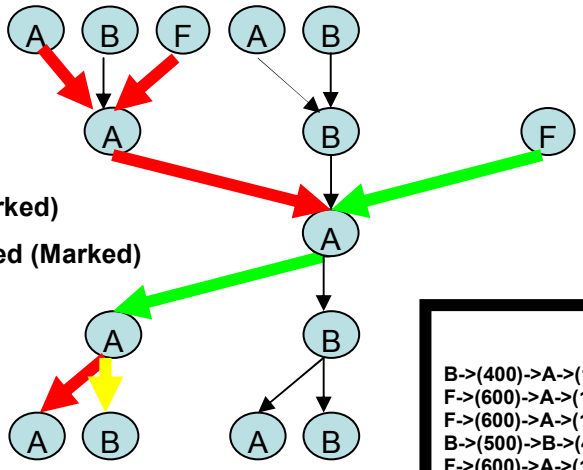
Query
Target: A
Length: 2

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B
B->(500)->B->(400)->A->(100)->A->(200)->B

F->(600)->A->(100)->A->(100)->A(SKIPPED)
(CENTER)

Print Path For ‘A’ And Length 2



- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

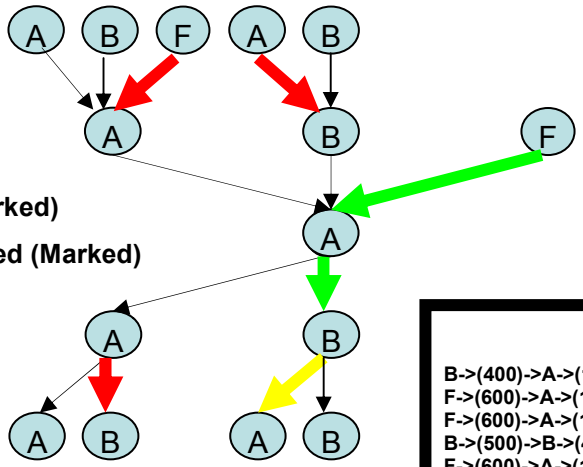
Query
Target: A
Length: 2

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B
B->(500)->B->(400)->A->(100)->A->(200)->B
F->(600)->A->(100)->A->(200)->B

F->(600)->A->(100)->A->(200)->B(PRINTED)
(CENTER)

Print Path For ‘A’ And Length 2



- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

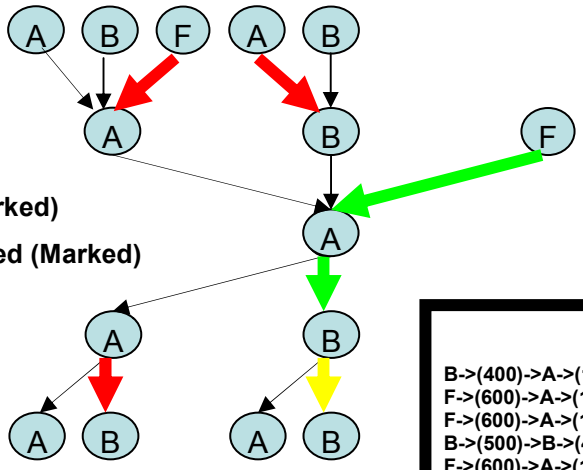
Query
Target: A
Length: 2

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B
B->(500)->B->(400)->A->(100)->A->(200)->B
F->(600)->A->(100)->A->(200)->B
F->(600)->A->(200)->B->(400)->A

F->(600)->A->(200)->B->(400)->A(PRINTED)
(CENTER)

Print Path For ‘A’ And Length 2



- Printed (Unmarked)
- Might Be Printed (Marked)
- Not Printed

Query
Target: A
Length: 2

SOLUTION

B->(400)->A->(100)->A->(200)->B->(500)->B
F->(600)->A->(100)->A->(200)->B->(400)->A
F->(600)->A->(100)->A->(200)->B->(500)->B
B->(500)->B->(400)->A->(100)->A->(200)->B
F->(600)->A->(100)->A->(200)->B
F->(600)->A->(200)->B->(400)->A
F->(600)->A->(200)->B->(500)->B

F->(600)->A->(200)->B->(500)->B(PRINTED)
(CENTER)

5.4 Apply Unigram and Association Score Cutoff

5.4.1 Unigram Cutoff.

Unigram cut is the minimum count that every unigram in the vocab file must possess in order to be included into the network. This value of unigram cut is thus used to modulate the size of network built in the system. The unigram cut is tricky in its own sense as it also means that all the entries in the bigram files and higher order n-gram cut is implicitly restructured based on the frequencies of the unigrams in the vocab file.

Where does this fit in the present implementation ?
In the present implementation of the system, the per unigram characteristics are stored in a structure. This structure is thus representative of features of every entry in the unigram. These characteristics include

1. storing the unigram count.
2. storing the linked list of edges that hold the information of edges that are incoming and outgoing from the node.
3. custom defined variable used for network analysis.

Thus, we declare an array of pointers to structures that are representative of every unigram built into the network. To implement unigram cut, we could approach it in the following two possible ways:

1. It makes a very good managerial sense from coding perspective to not to disturb the existing code that builds the network and write and additional module quite discreet from the network building module that prunes the array of structures of all unigrams whose frequencies are below the unigram-cut.

2. Otherwise, an innocent answer to it could be never to have unigrams in the array of structures whose frequencies are below the unigram-cut. This approach is very attractive as it does not effect the functionality of subsequent modules written

before but it is very difficult to divide the code into discreet modules.

Advantages Of First Approach:

1. The modules are discreet. We could identify each module visually independent of other.
2. We know the unigrams that are left out in the network building stage. Thus, we have the list of all unigrams and the unigrams that satisfy the cut-off condition.

Disadvantages of First Approach:

1. The presence of a unigram with null as many of its characteristics is going to effect other modules following the unigram cut module as these modules are not designed to handle null in their fields.

2. There is memory wastage due to allocation of space for unigram in the array of structures, which would undermine the very reason for presence of unigram-cut.

3. Presence of unigrams that does not "satisfy" unigram cut does not yield anything beneficial to us. While one could argue that the comprehensive knowledge of unigrams may be useful in the future, it does not make logical sense of including it in the use of unigram-cut in the first place. Keeping the knowledge of all unigrams in the vocab file can be implemented in a separate module.

   The purpose of using unigram cut to carve the network to our wish is to include only the unigrams that lie above the a given cut-off limit and not have any unigrams that do not satisfy the cut-off condition.

Advantages of second approach:

1. The memory requirement is cut down by huge proportions. An example is that the

number of unigrams above the frequency of 200 are to the order of 13 million, the number of unigrams above the frequency of 500 is to the order of 69,000. This is a very effective tool in pruning and carving the graph to our requirements by saving memory.

2. The modules using the structure built, need not worry about any side effects due to use of unigram-cut as it is just the size of unigram array that is scaled down.
3. The time taken for further processing is greatly cut down.

Disadvantages of Second approach:

1. The only disadvantage of the second approach is that implementing this method involves weaving in and out of network building code. There are several conditions that need be checked for deciding for the size of the unigram array and inclusion in the bigram array.
2. Clear delineation is not possible. This functionality does not come under integration, but the very characteristics of the second approach requires to use a very conditions carefully in the network building code and check for unigram cut-off conditions.

Approach taken for the project:

The project uses the second approach to for unigram – cut because of its inherent advantages. The logical steps followed for the unigram cut are:

1. Initialization of the size of the unigram array.

   The size of the unigram array is initialized with the number of unigrams that are above the unigram cut-off. To implement this, the vocab file is in its entirety and is checked for number of words that are greater than the cut-off frequency. This number is read and size of the array is initialized.

2. Building of the unigrams into the array:

After initializing the array to appropriate size, the unigrams are initialized as they are read from the vocab file.

3. Building the network of Bigrams:
   The addition of bigrams to the network is done after it is been verified that the two of the unigrams that make up the bigram are present in the unigram array. This is verified from the binary search. If the binary search return -1, it means that the unigram is not present in the unigram array.

Verification:

1. The code was run over a test data to verify its correctness. The results obtained matched the one in the test data.
2. A total count equal to the number of unigram in the vocab file when the unigram cut is set to 200.

5.4.2 Association-Cut:

Association cut is float point number between 0 and 1. Association cut is the ratio of bigram frequency to the product of individual unigram frequencies. While paths in the graph are being searched, a given bigram is checked if its frequency is above the association cut. If the bigram frequency is not satisfied, the whole of the tree following the bigram is omitted. Thus, association cut is also a tool to carve the graph we according to a parameter.

5.4.3 Graph Analysis (Deprecated)

This is an alternate approach to network analysis implemented in the project. It is different from the one in the project in the following ways:

1. It works in master-slave mode.
2. The statistics of the network are calculated in an incremental fashion. It simply means that the characteristics of one single network are calculated first before proceeding to the next one.
3. It is based heavily on message-passing model and broadcast analogy.

4. It is implementation of depth first search in parallelized fashion.

Algorithm:

1. The processor 0 is assigned the status of master and all the other processors are assigned as slaves.
2. Running the whole process can be very confusing to code and debug. Therefore, I have used the following approach.
3. The slaves are always in receiving mode. They receive the commands from the master and perform tasks accordingly,

   For example:

   the following numbers acts as instructions to the slaves:

-4 : perform a local check in its graph local to the processor, for the occurrence of word received in previous message.

-5: Send a signal to the master if a new node is found in its sub-network.

-2: Slave is assigned control to broadcast the new nodes found in its sub-network to all other processors.

-3: End of the program, every slave exits from the waiting state and finalizes the control (i.e reaches MPI_Finalize).

If the message received from the master are none of the above, then it is assumed that the an index from the master is received that corresponds to the index to searched in the local network.

Thus, this model can be viewed as a master-slave network, where the master instructs the slave to the task to perform next and thus we are controlling every executing task of the slave.

With this understanding the algorithm proceeds as follows:

1. The slave selects the first token in the unigram array and marks it to as belonging to a network one. It performs a depth first search on the word in its own localized sub-network.
2. Since, the network is distributed among processors, every word for which a depth-first search is performed, it is broad casted to the slaves along with the network id.
3. The slaves receive the index of the word that needs to searched and the network id and performs a localized Depth first search in its own network.
4. It appears at this point that the whole of the network is traversed. But, consider the following scenario.

Processor 0 has the edge A->B.
Processor 1 has the edge C->D.
Processor 2 has the edge B->C.

Thus, if processor 0 is the master , it broadcasts A and B.
Processor 1 checks for the occurrence of A or B but finds none.
Processor 2 has the edge B->C, therefore, this edge is discovered. However, C->D is not discovered.

Thus, broadcasting from the master alone is not the solution. This requires that the slaves need to broadcast their found tokens too.

5. The next step would be to broadcast the found tokens in the all of the slaves to every other processor so as to reflect the discovery of the new tokens in the slaves.

6. The master then checks its own copy of the unigrams to check if any new nodes have been found by the slaves.
7. If new nodes were found in the slaves, then the process is repeated from step 2

with the newly found nodes.

8. If no new nodes were found, it means that the network has been completely discovered.
9. After a single network is completely discovered, the next unmarked node is selected and the algorithm is repeated again from step 2.

This approach has been tested for test data and is found to work correctly.

Advantage Of this Approach:

1. This approach is proceeds one step after next. i.e the network detection takes place incrementally.
2. To find out the network for which the word belongs to, this approach could be the easiest and light weight to implement.

Disadvantages:

1. The method involves a lot communication between the master and the slave.
2. This overhead due to message passing is immense.
3. Due to large overhead of message passing, this method is ideally suited for bigram files with large number of small networks.

Future Work:

1. The future work on this method could be an intelligent use of this algorithm in finding a network specific feature. More clearly, for a scenario, where the statistics of a single network is important but not the information about all the networks, this algorithm is ideally suitable.
2. In the current implementation, the overhead due to message passing could be reduced by substantial amount if a list of nodes that are already discovered in each of the processors are maintained for each iteration when the network is discovered to its entirety. Thus, maintainence of such list of nodes avoids broadcasts of the same node in every iteration.

6. EXPERIMENTS AND EVALUATION

6.1 Testing Methods

6.1.1 Testing methods for network building

(Author: Darshan Paranjape)

As we are dealing with a very large amount of data, it is very difficult to test whether created network carries the correct information for every node. For testing purpose, we have created sample data files for both unigrams & bi-grams. Unigram sample file consists of 40 unigrams with varying frequencies. Bi-gram sample file has 51 entries. The bi-gram information is derived in such a way that there are 5 distinct networks. So the same sample file can also be used to while testing the network analysis code that finds total number of disjoint networks.

We have written a function named 'display' that prints the information of each node in the network. This information includes token associated with the node, number of outgoing edges, list of all outgoing edges & their weights, number of incoming edges, list of all incoming edges & their weights. When this function is called by every processor, we can see the node information in whole network. The main testing method used while building the co-occurrence network is to cross-check the output of the display function with sample data files.

It is not possible to test the network built on Google N-gram data with 'display' function. But there are several other checks that could be performed. We have cross-checked the number of entries read from unigram & all of the bi-gram files. We have manually cross-checked the index returned by 'binary search' by comparing it with unigram file entry.
We also displayed the outgoing edge information of several randomly picked nodes from all processors. This information was crosschecked manually with entries of these nodes from Bi-gram files.

6.1.2 Testing method for Network Analysis

(Author: Bin Lan)

For network analysis, we manually generate several testing samples which consists a lot of different network layouts, including cycle, total isolated nodes, self loop, incoming edge/outgoing edge only. Also, we randomly generate some testing samples with no specific network layout. When we test with Google uni-gram and bi-gram data without any cutoff, the algorithm terminated and displays network information with expected layout. However, it is impossible to prove all the outputs are correct since some of the sub-networks consist of a very large amount of nodes and edges. So we pick some of the small sub-networks and compare with the data from n-gram files. They all show matched results. For example, both our program and Google n-gram data file show that "106Test" is a self-loop network.

6.1.2 Testing method for path finding

(Author: Anurag Jain)

1) We tested the code on a unigram file with three elements and a bi-gram file with five elements. The two files are the used as example. It worked well with it without any problems.

   We used blade for this of testing.

   We were basically looking for the following things here:-

   a) The results are perfect.
   b) The printing results are in the required format.
   c) If the word doesn't occur in the unigram then we print its non-existence.
   d) Cycles are not printed.
   e) Complete paths are printed.
   f) The search is case sensitive or not.
   g) Timing information is printed separately from the network being built.

2) Then we started testing on the whole network. It worked fine for some time and then gave timeout error. After long time of testing we realized what the problem was. The memory resources available with the 0^{th} processor were limited and cannot support the whole network therefore that node was unexpectedly exiting giving the timeout error.

Issues

Timeout Error while running test runs on big data. Execution starts and results start printing but after some time the timeout error unexpected occurs.

Hints

The error file suggested the Singal 9 error. Node 0 exited unexpectedly.

Observation

The timeout error occurrence is delayed if we increase pmem from 2 GB to 4 GB. So we were sure memory is the problem this time. So finally we tried with pmem as 7 GB memory with 32, 64 processors we found we were short of our memory

6.2 Performance Evaluation and Benchmarking

6.2.1 Performance for Network Creation

(Author: Darshan Paranjape)

We have designed the network creation algorithm in such a way that minimizes the file I/O operations. After building the network, the node information remains in the memory. We are not storing the network outside memory. The main reason behind this purpose is to avoid time consumed in file I/O. By keeping the network information in the memory, make the network creation step faster. This method poses no problems while getting the output of the alpha stage i.e. while finding the disjoint networks. In fact it is much easier & faster to access the network without involving file I/O again. But this method also takes toll on the available memory. As we are making use of unigrams, we have to maintain node structure array of size 13 million approx. In addition to this, we are storing the information from Bi-gram files in outgoing & incoming linked list. Maintaining the linked list of edge structure also requires a lot of memory. This is where tradeoff between memory and speed comes along. If we want to use minimal memory, we'll have to store the network outside memory & thus will have to introduce file I/O which in turn will slow down the program considerably. If we want faster program, we'll eliminate file I/O eating a lot of memory. So far, we've gone with approach that gives us faster program. But the approach can be changed if necessary.

Memory Analysis:

We are allocating memory mainly to the 2 structures used in the program structure "node" and structure "edge". We are maintaining array of node structures. As this array is based on the unigram data, its size is equal to number of unigrams i.e. 13588391. While reading the unigram file, we dynamically allocate some memory to store unigram in the token field of node structure based of length of unigram. This is much better method than

statically deciding array size because max length of unigram is 40 but average length cannot be more than 15 in length. Each array element is of size 88 bytes plus 15 bytes (average) for token. So size of 1 array element = 103 bytes. There are 13588391 array elements.

So total memory required for node structure array = 13588391*103 bytes
= 1399604273 bytes
Total memory required for node structure array = 1.3 GB

Size of the edge structure is 32 bytes. For each entry in the bi-gram file we create 2 edge structure elements, one for the incoming linked list & other for the outgoing linked list. First 31 bi-grams have 10000000 entries whereas last file has 4843401 entries. But these entries are (almost) equally divided in available number of processors.

So total memory required for edge structure linked list by p processors
= (10000000 * 31 + 4843401) * 2 * 32 bytes
= 314843401 * 64 bytes
= 18 GB
This is the memory required for p processors.

So to store edge information, each processor needs 18/p GB memory.

Total memory requirement for each processors = (18/p) + 1.3 GB
Where p = total number of processors

So minimum system resource requirement for network creation stage is
Number of processors = 4 with following configuration
pmem =7 GB, nodes=4: ppn=1

Recommended system resource requirement are
Number of processors = 32 with following configuration
pmem =3 GB, nodes=16: ppn=2

Each node has 7GB memory that can be used. If we use 4 processors per node, this 7GB memory is divided in 4 so each processor gets 7/4=1.75 GB. So it is recommended to use only 2 processors per node rather than using 4 to avoid running out of memory.

Benchmarking:

Following is the benchmarking information for building the co-occurrence network on all 32 files of Bi-gram data. It provides information about the number of processors used for network creation, memory requirement for each processor and time required to build the network. Timing information given is captured using MPI_Wtime function.

Number of Bi-gram Files	Number of Processors Used	Memory Required per Processor	Time required to build the Network
32	4	5.8 GB	864.9280 Seconds
32	8	3.55 GB	478.24972 Seconds
32	16	2.425 GB	325.36590 Seconds
32	32	1.8625 GB	236.64958 Seconds

6.2.2 Performance for path finding

(Author: Anurag Jain)

Memory Analysis:

Memory requirement for 0^{th} processor = 25 GB (0th processor)

Memory requirement for each processors except processor 0 = ((16.5/p) + 1.5) (p-1) GB

Total Memory Requirement = 25 GB + ((16.5/p) + 1.5) (p-1) GB

Where p = total number of processors

Preffered System: Altix 3700 BX2

Why SGI Altix?

Processor 0 is short of memory on blade. The blade-center provides 7 GB of memory per node. So even if we used 1 processor per node (4 processors can be used per node) we had a maximum of 7 GB available per node which according to calculation was less if in the worst case the whole network resides on processor zero. The system might work for some target words with or without cutoff and small lengths on IBM Blade Center but the real power of the system can only be realized on Altix 3700 BX2 because of the tremendous memory requirement for path printing. So to run without Signal 9 errors because of lack of memory Altix is the system to go for.

Minimum system resource requirement for the whole system:

Number of processors = 4 with following configuration

mem = 42GB, nodes=1: ppn=4

Recommended system resource requirement for the whole system:

Number of processors = 4 with following configuration

pmem = 50Gb, nodes=1: ppn=4

Benchmarking Results

Altix 3700 BX2

Input Token: fountain

Mem: 50gb

Cutoff = 0

Number Of Processors	Specified Length	Time Taken To Print Query(Seconds)
4	4	Job Blocked (File Size > 48 Gb)
4	3	Job Blocked (File Size > 48 Gb)
4	2	Job Blocked (File Size > 48 Gb)
64	1	233.876060

IBM Blade Center

Input Token: fountain

pmem: 7gb

Cutoff = 1000

Number Of Processors	Specified Length	Time Taken To Print Query(Seconds)
4	2	2985.600088

Input File: target

pmem: 7Gb

Unigram Cutoff = 100000

Associative Cutoff = 0.000000009

target File

Nike 1

thrilling 2

wonderful 2

miserable 1

tragic 4

Bush 1

Duluth 4

duluth 4

abczzzddd 4

Number Of Processors	Time Taken To Print Query(Seconds)
4	0.103744
8	0.159413

Remark On Benchmarking Results

Cutoff plays an important role in reducing the size of the overall network which allows us to print more specific paths. If we increase the cut-off to a very high value the number of paths printed is reduced tremendously.

The time taken is increased with increase in the number of processors because the network is even more distributed. So, collecting back the network takes more time.

IMPORTANT OBSERVATIONS

Paths And Unigram Cut-Off

Unigram Cut-Off is the heart. Unigram Cut-Off helps us to reduce the size of the network the much we want and that helps us in running our system even with less amount of memory.

Paths and Associative Cut-Off

If Unigram Cut-Off is the heart then Associative Cut-Off is the soul. It helps us in running the system with high lengths even with lack of disk space. No only this it also helps us in reducing the size of the collected network.

Ques: Which system is suitable for printing?
Ans: We can use IBM BladeCenter or SGI Altix 3700 BX2.

IBM BladeCenter

It can be used only with high unigram cut-off''s and very small associative cut-off''s due to lack of memory and disk space respectively.

SGI Altix 3700 BX2

It can be used even with no unigram cut-off''s and very small associative cut-off''s due to availability of loads of memory but limited disk space respectively.

NO PATHS PRINTED FOR LENGTH GREATER THAN HALF THE NUMBER OF EDGES IN THE DISJOINT NETWORK

The reason for that is because we avoiding cyclic loops we cannot print paths with edges more then the number of edges in the disjoint network as it will result in cycles.

7.CONCLUSION AND FUTURE WORK

In this project, we have presented a parallel method for building the co-occurrence network based on Google n-gram data and discover paths for a given target word. Overall, we have shown that parallel approach has a lot of advantages over sequential program.

To continue improve our method, we would like to do few things in the future. First, a more organized way to distribute the edge information would benefit the rest of our functions. Now the edges are distributed based on data decomposition. In the future, we would like to distribute the edges based on the end points it connects to. Second, a faster algorithm for analyze the network would speed up the whole process. Right now, the algorithm works fine but cannot utilize the number of CPUs we are using. Also, we realize that discover path(s) for a target word does not strongly connect to analyze network. So we could either parallelize both functions or made the analyze network function an optional part of our system.

BIBLIOGRAPHY

1. Niwa, Yoshiki. , Nitta, Yoshihiko., "Co-Occurrence vectors from corpora vs. distance vectors from dictionaries" Advanced Research laboratory, Hitachi Ltd, Japan.
URL: http://acl.ldc.upenn.edu/C/C94/C94-1049.pdf

2. Veling, Anne. , Weerd, Peter van der., "Conceptual Grouping in Word Co-Occurrence Networks" Medialab, The Netherlands.
URL: http://dli.iiit.ac.in/ijcai/IJCAI-99%20VOL-2/PDF/006.pdf

3. Cohen, AM. , Hersh, WR. , Dubay, C., Spackman, K. "Using co-occurrence network structure to extract synonymous gene and protein names from MEDLINE abstracts" Oregon Health & Science University, USA
URL: http://davinci.ohsu.edu/~cohenaa/Cohen2005-GeneSynonyms-BMCBioinformaticsProvisional.pdf

4. Edmonds, P., "Choosing the Word Most Typical in Contest Using a Lexical Co-occurrence Network" University of Toronto, Canada
URL: http://acl.ldc.upenn.edu/P/P97/P97-1067.pdf

5. Ferret, O. "Discovering word senses from a network of lexical co-occurrences" CEA-LIST/LIC2M

URL: http://acl.ldc.upenn.edu/C/C04/C04-1194.pdf