What A Tangled Web We Weave; The Huge Net Of Words – Google N-gram Data analyzer

Team Members:

Anagha Dharasurkar

Andrew Norgren

Premchand Bellamkonda

Salil Bapat

Shruti Pandey

20th Nov 2007

Department of Computer Science

Course: cs8621-Advanced Computer Architecture

Professor: Dr. Ted Pederson

Project Webpage: http://google-ngram.sourceforge.net/

1.0 Problem Definition [Andrew Norgren]

The project is to build and use a co-occurrence network from the Google N-gram Data [1]. The project will be completed in 3 stages. The goal of the alpha stage was to build the network and find several network statistics. The goal of the beta stage is to access the network, or as much of it as you can. Also the network building should be filtered and words with less than a certain cutoff frequency should not be included. In the final stage we have to include an associative cutoff for pair of words. The project is implemented in C and MPI is used for achieving parallelism.

The experimental data that used is the “Google N-gram Data” which consists of one trillion words that has been collected by Google Research. Google Research uses n-gram models for various research and development projects. An n-gram model is a model that takes the words found from public Web pages that occur together, and keeps track of the frequency (or how many times those occurrences are found.) The “n” denotes how many words that occur together. For example, a bi-gram model finds the frequencies of pairs of words that occur together. It is believed that this amount of data will be able benefit the research community in multiple ways, so Google Research decided to share this data. They distributed the data on six DVDS containing 24 GB of compressed text files. Here is an example of what the data looks like:

(This is 2-gram data)

```
collectables collectibles 55
collectables fine 130
collected by 52
collectible pottery 50
collectibles cooking 45
```

ceramics collection 144

ceramics collection 247

Here are the statistics of the data:

Number of tokens:	1,024,908,267,229
Number of sentences:	95,119,665,584
Number of unigrams:	13,588,391
Number of bigrams:	314,843,401
Number of trigrams:	977,069,902
Number of fourgrams:	1,313,818,354
Number of fivegrams:	1,176,470,663

The unique words are words that appear at least 200 times. and the five-word sequences are sequences that appear at least 40 times. Google Research is interested to hear back from anyone who used this data in any way.

In this project we are mainly dealing with the bigram data. We will also be using unigram data for some of the tasks. We are not going to deal with 3-gram or 4-gram or 5-gram data provided by the google.

2.0 Co-occurrence networks

2.1 Introduction [Salil Bapat]

A co-occurrence network links together words that have occurred together in some piece of writing. Each word represents a node in the network, and the edges between the words indicate that they have occurred together. Note that the edges are directed, and show the order in which the words have occurred. The weights on the edges are the frequency of the two words occurring together in that order.

For this project, we are going build co-occurrence networks of the bigram data provided by Google. Here is an example of the co-occurrence network for the bigram data.

Consider the following bigram data:

the match 100

match is 200

is a 1111

is of 2112

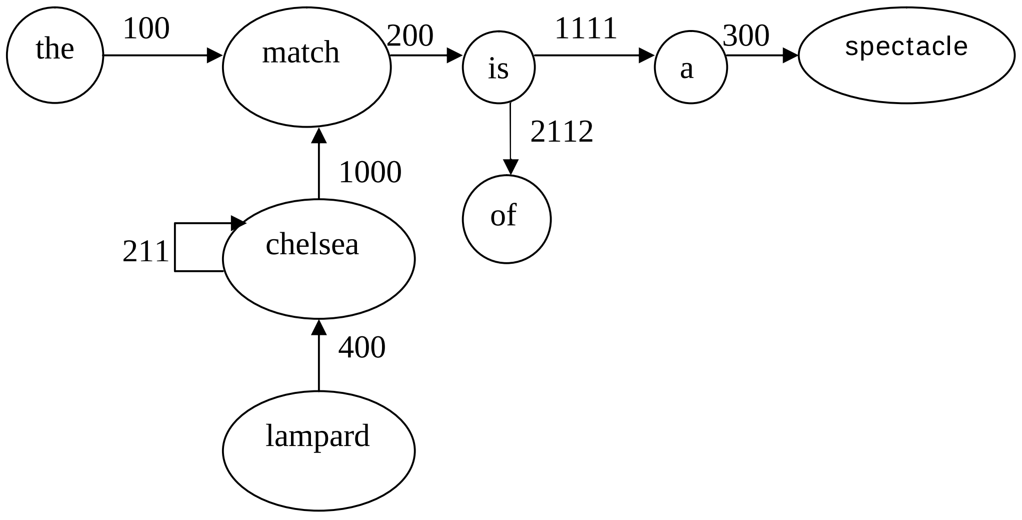
a spectacle 300

chelsea match 1000

chelsea Chelsea 211

lampard chelsea 400

Then the co-occurrence network will look something like this:



As you can see, each bigram is represented by 2 nodes and an edge connecting the 2 nodes. The edge is a directed edge and it starts from the 1st bigram and goes to the 2nd bigram. The value of edge represents the frequency of occurrence of the bigram. As you can see there can also be self loops in the co-occurrence networks. In the above example, the bigram “chelsea chelsea” occurs 211 times in the dataset. Hence you can see a self-loop on the node representing “chelsea”. The weight of self-loop is 211. Self loops do occur in the google-bigram data.

Co-occurrence networks can be represented and used in several other ways. E.g. they can be used to represent distances between locations, chemical reactions, mathematical conversions etc. Following are some of the previous works being done using the co-occurrence networks.

2.2 Co-occurrence vectors from corpora vs. distance vectors from dictionaries [Andrew Norgren]

A comparison and description of co-occurrence vectors and distance vectors is given in the paper, “Co-occurrence Vectors from Corpora vs. Distance Vectors from Dictionaries.” Word vectors are developed and used to provide word meanings through numerical approaches to semantics. Co-occurrence statistics require larges sets of data that include rare words. This paper describes a method developed that constructs word vectors from a dictionary that handles rare words easily. These are called distance vectors. A distance vector is made from a reference network, which is a network of the words that are used to define a word. From these reference networks, you get word vectors, which are the lists of distances from a particular word to selected words, called origins. The distance vector is the distance from a word to these origin words. Co-occurrence vectors are the lists of the co-occurrence likelihoods of a word with the selected origin words. The cooccurrence likelihood is a function derived using densities and probabilities between two words. Distance vectors and

Co-occurrence vectors were used in two experiments to compare two different semantic tasks. The first experiment was “Word Sense Disambiguation”, which is the problem of determining what sense, or meaning out of a word’s multiple meanings, a word has in a given sentence. The method this paper used to determine this was to use context vectors, which is the sum of the constituent word vectors. The results showed that using co-occurrence vectors produces higher precision than distance vectors, which means co-occurrence vectors are generally better. The second experiment was “Learning of positive or negative meanings,” which is to determine if a sentence has a negative or positive meaning. These results showed that distance vectors have a higher precision, showing that positive or negative meanings are shown better in distance vectors than in co-occurrence vectors. [5]

2.3 Choosing the Word Most Typical in Context Using a Lexical Co-occurrence Network [Premchand Bellamkonda]

This paper explained about a partial solution to the problem of lexical choice by using cooccurrence network. It explained about a new statistical approach to modeling context that helps in determining the most typical, or expected synonym in a given context. Here we might have to consider the context beyond the surrounding of the context to achieve satisfactory results. By doing so we will have greater power to assume the occurrence of that word and weight the word according to its information content. Here even though the word does not co-occur with another given word, but it will help to predict the use of that word if these two words are mutually related to third word. When building the network, root is connected to all the words that co-occur with it in the training corpus and again connect these words with their co-occurring words up to some specific depth. We use mutual information scores and t-score to determine a co-occurrence network. For choosing the most typical word, we find the sum of the scores of each co-occurrence of the candidate with a word in the sentence. To evaluate, the author took two tables that has the data about the frequency of the words. The author could not come to prefect conclusion by table 1 as different patterns of words can occur. But from Table 2 he found out that second-order results were helpful to achieve results better than baseline accuracy. By this we can say that we need more than surrounding context to build and adequate contextual representations. [3]

2.4 Discovering word senses from a network of lexical co-occurrences [Premchand Bellamkonda]

This article presented a possible solution to find the nature of senses of a word from a network of lexical co-occurrences. It also differentiated the senses of the word from the network. Semantic resources have proven to be useful in information retrieval but only if word sense disambiguation is performed with great accuracy. Semantic networks were criticized for incompleteness and their relations with synonyms. An algorithm was designed to find the word senses and is done word by word in which processing of the word relies in the sub graph that contains co-occurrence. A network was build for English and French such that we got total number of different words and their occurrence in the corpus. Word sense discovery algorithm is explained.
In building the similarity matrix between co-occurrent, if there is no relation between two words in the network, similarity is noted as zero. This makes the network to build easily and is also efficient. But some semantics are difficult to catch. In SNN algorithm, clustering is performed by detecting the high-density areas of a similarity graph.
When the experiment the algorithm with corpus of English and French words, we observed that significant amount of words do not have any sense, even in second order occurrence. For evaluation of results, we compare the results with a standard resource that is

acknowledged as golden standard. We got a satisfiable result of 88% agreement between its result and human judgments. So, the results obtained in first order co-occurrences, which produces a higher number of senses by word, are lower than the results obtained with second order occurrences. [4]

2.5 Conceptual grouping in word co-occurrence networks [PremchandBellamkonda]

This paper deals with query processing in large databases. Information Retrieval queries can often result in large number of documents found to be relevant which are sorted by relevance but not by analysis. If the retrieved information is of several pages, then it might be hard for the user to retrieve the specific query. So here the author expalined about the conceptual grouping which will try to find the exact meaning of the user query in a document collection. In building the cooccurence network, we consider the words to be co-occurent only if they appear in document d and they are less than 50 words apart.
We find the total number of co-occurence networks in the whole document. The relevance of each co-occurent network is also found by using the distance between the words and the probablity of the occurence of the word. We summarize all the documents and find the strength between all the words i and j. For every word i, we find the 80 best words j that co-occur with i. In the conceptual grouping, all the words linked to the query int the co-ocurence network are likely to be semantically related to the query. By using grouping algorithm we will be able to present the best three or four words for each group. By using conceptual grouping, we can give the user an overview of what he is interested in. After the user types in the query, all the documents titles that contain the query words are returned. The user can browse through these documents and view the original documents in another frame. Testing shows that this technique can improve precision, helps user find what they need more easily and give them a semantic overview of the document collection. [6]

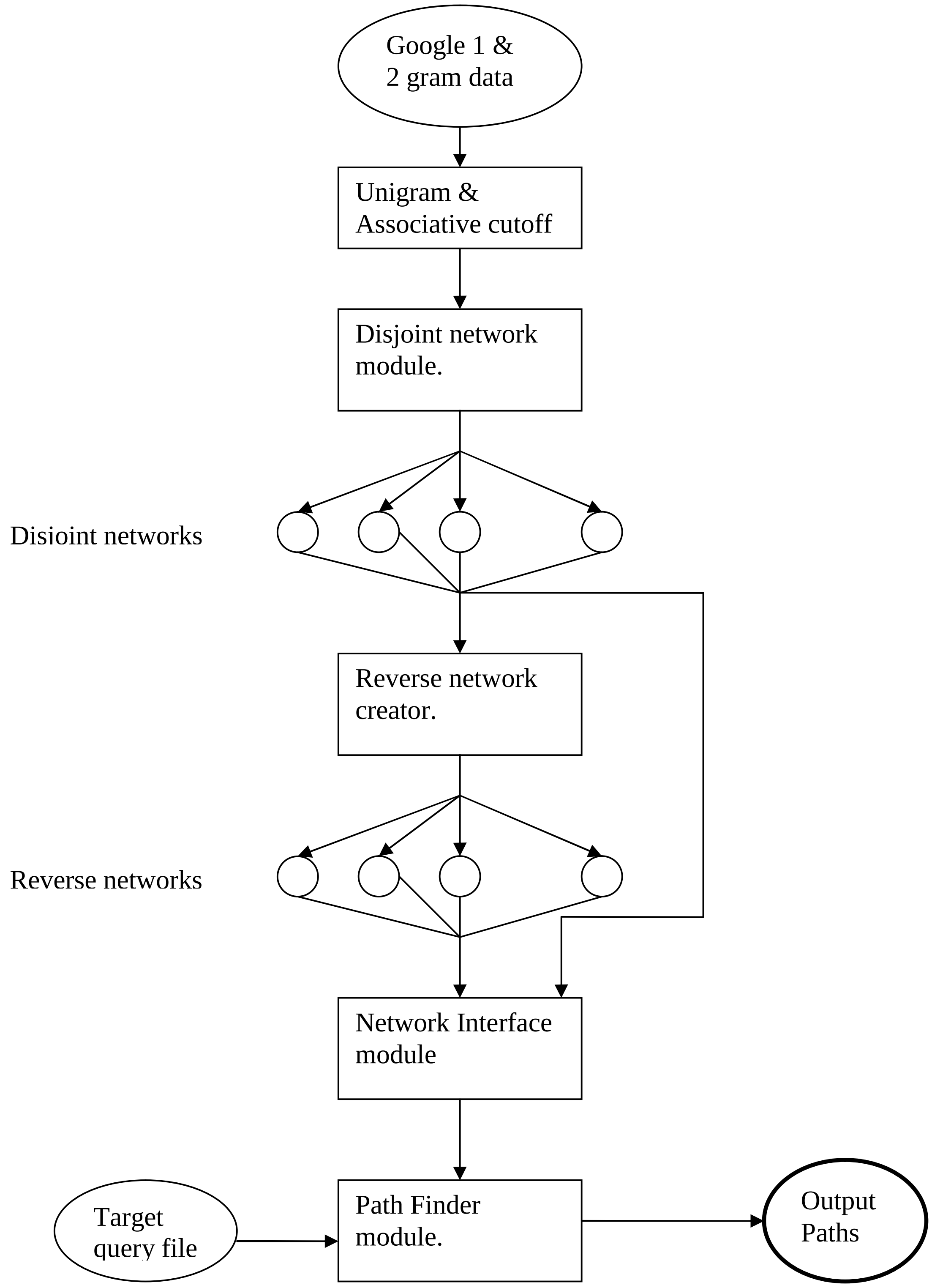
2.6 Incremental Construction of an Associative Network from a Corpus [Premchand Bellamkonda]

The paper discusses about a computational incremental network model which is aimed at modeling the development of human semantic memory. The authors discuss six main features namely: the kind of input, knowledge representation, way of adding new context, the unit where the co-occurrence is present, presence higher order co-occurrences and compositionality, which are present across the various existing models. These existing models include Latent Semantic Analysis (LSA), Hyperspace Analogue to Language (HAL), Sparse Random Context Representation (SRCR) and Word Association Space (WAS).
The authors provide their model ICAN in the paper. ICAN is based on network representation, which is more accurate. Each word and its neighbor are associated with weights varying between 0 and 1. The model is an incremental model which allows addition of cooccurrences and deletion of neighbors if the strength of co-occurrence becomes weak. The model uses a window to capture the words and the co-occurrences with an option to vary the window size based on the accuracy required. The tests were run with a French child corpus and the results were satisfactory. Using a window size of 11, co-occurrence effect p□□p + (1-p)/2, p □□q + 0.02 (1-q) (p.m) and occurrence without co-occurrence effect p □□0.9 p, the similarity values at the 1st associate level was 0.415, which was more than other models.
The authors accept that the model has some initial blips but the initial results are encouraging enough to display the model's potential. The model would simulate the human word association pattern more closely than the vector representation methods. [2]

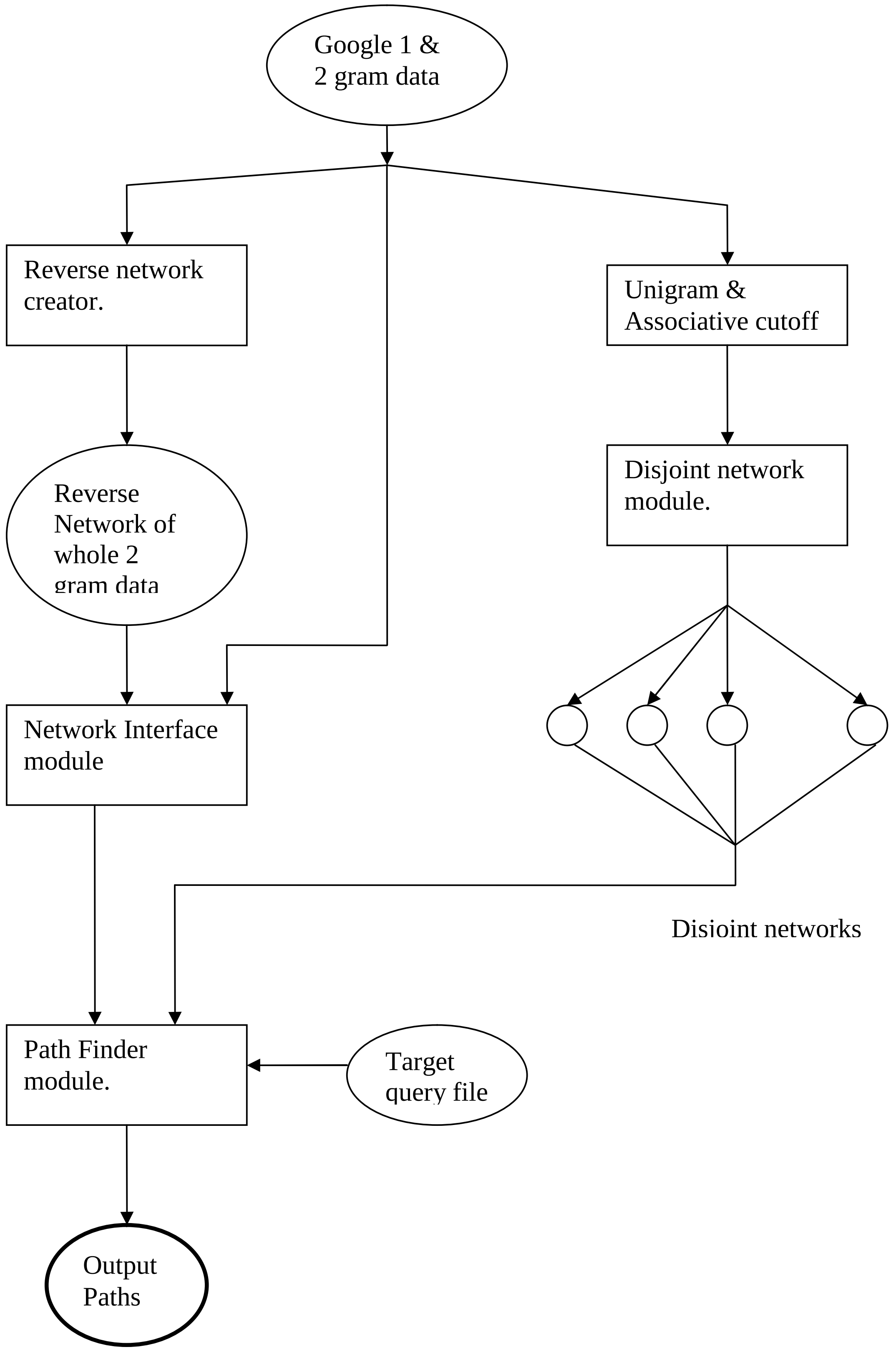
3.0 System Architecture [Salil Bapat]

The diagrams below show the architecture diagrams of the system which we had planned and also the system which we ended up implementing. Diagram 3.1 is the architecture that we had planned. In that the google 2-gm data will be first accessed, then the words which do not satisfy the cutoff conditions will be removed. After that the disjoint networks will be created and stored in files. Of these disjoint networks, reverse networks will be created. Then the disjoint networks and the reverse networks will be used to find query paths. However due to several implementation problems, we did not get enough time at the end to do put our components in the architecture we had planned. Hence we ended up implementing a slightly different architecture as shown in Diagram 3.2. Here the disjoint network finding and the reverse network creator module are not integrated. So the disjoint network module will find the disjoint networks and other network statistics and the reverse network creator will create the reverse network of entire google 2 gram data. The path finder module will use the google 2-gram data and the reverse network to find query paths. Due to this amended architecture, we were not able to put all the individual components in proper architecture to make complete use of the unigram and associative cutoffs.

3.1 Planned Architecture



3.2 *Implemented Architecture*



4.0 Co-occurrence network building [Salil Bapat]

4.1 Network building background

Before building the network I did some analysis on the existing google 2gram data. The data is distributed in 32 files. The file names start from 2gm-0000 to 2gm-0031. The data is stored in these files in sorted order. Along with these files, there is an information file, 2gm.idx, specifying the start token in each file. Thus the 32 files themselves act as a co-occurrence with the 2gm.idx acting as an hash for the information. The beta stage involves finding all the "from" and "to" links of a given node. On doing some analysis I found that the time required to access one of the 32 files and find all the "from" links of a node is quite fast (e.g. in the worst case scenario, time taken to search a file with 10000000 bigrams is approx 4 secs). Hence to find the "from" links, using the existing structure is the best approach. Problem comes when you need to find all the "to" links of a node. Since the data sorting is done based on the "from" links, there is no quick and easy way to find the "to" links data from the given set of files. Hence the network I built gives a very quick access to only the "to" links of a particular node. Hence the name of the root-network directory is also called google-ngram-reverse-network. So in the beta stage, if we want to find all the from links of a word say "abc", then we will first go in the 2gm.idx and get the file name which has all the bigrams of "abc". This file will then be accessed to get all the bigrams. On the other hand, when all the "to" links of the word "abc" will be required, the network which I build will be accessed to get all the connected nodes. A completely different approach would have been to build a complete network and keep it in memory itself. However, this would have meant that the network had to build everytime a query is run.

4.2 Network Details

The network will be stored in a directory structure with root directory as "google-ngram-reversenetwork". This parent directory contains 63 pre-defined directories. 26 directories of alphabets "a" to "z", 26 of alphabets "A" to "Z", 10 of numbers 0 to 9. And an additional directory named "symbols". These 63 directories will be created right at the start of the program. After these are created, we start reading the bigram data. For each bigram found, another level of directories will be created as follows:
If there is a bigram
over get 1000
a directory will be created under the directory google-ngram-reverse-network/g/, named ge. So there will be a structure created like google-ngram-reverse-network/g/ge. Inside this directory a file named to_Corinthians will be opened in append mode. In this file the bigram entry will be stored in the reverse order. So the bigram will be stored as:
get over 1000.
Similar processing will be done for all the bigrams. So for e.g., the file which we created above will contain all the bigrams who’s second token starts from "ge". If the bigram starts with a symbol (non-alphabet non-number), then it will be stored in to_Corinthians file inside the directory googlengram-reverse-network/symbols. By having this 3-level directory structure, the data access is much faster and simpler. This structure will be used to find all the "to" links of a word in beta stage. So, if we want all the "to" links of the word "get", then we will first open the directory google-ngram-reverse-network/g/ge, then open the to_Corinthians file inside that directory, then traverse this file and get all the bigrams starting with "get". Thus it will be really easy and fast to find all the "to" links of a word.

4.3 Parallelism Details

There were a few approaches that I evaluated for achieving parallelism. The first approach I thought of was involved the 0th processor reading all the bigram data. It will store this data in a dynamically allocated array of structures. It will then calculate the number of bigrams that each processor has to process, then send this number of bigrams to that processor. This was quite straight forward method. However, this involved too much of data transfer between processors and that reduced the efficiency of the program. The 2nd approach which I evaluated involved distributing files per processors. So if there are 8 processors, then each processor will get 4 files to process. If there are 9 processors then each processor gets 4 files, and the one remaining processor will work on one of the 32 files. However, with this, the efficiency increases only if the number of processors increases in the power of 2. The third approach which I evaluated (and finalized), eliminates the need to send huge data across and also the efficiency increases with increase in the number of processors. In this approach, first the processor 0 counts the number of lines in all the files. From this count and from the number of processors, it calculates the number lines which one processor should process. Starting from the 1st file and 1st processor, it sends the start position, the file number and the number of lines of each processor to each processor. (This is possible because the processor 0 also calculates the number of lines in each file along with the total number of lines in all files.) Then each processor, opens its file, skips first "start" lines, and then starts processing. If the file ends and if number of lines it processor has to read is still not reached then the next file is opened and read. All this process is possible because of ordered name of files starting from 2gm-0000 to 2gm-0031. I am using only MPI to achieve parallelism. I could have combined MPI and OpenMP, with OpenMP working on each processor and spawning 4 threads. However, the problem is that at any stage my input data is from a file which has to be read sequentially. Hence I cannot do any threading while reading and processing the file. Hence I was not able to use OpenMP. One problem with this whole approach is that there will be situations when several processors will try to write to the same to_Corinthians file. For e.g, processor 0 has the bigram "hello people 100" and processor 1 has the bigram "more people 200", and both these bigrams are evaluated at the same time, then multiple processors will try to write to the same file at the same time creating race conditions. I am avoiding this by creating one to_Corinthians file per processor. So instead naming the file to_Corinthians, its name will be to_Corinthians_(processor id). This will not cause any kind of bottleneck in beta stage. This is because in the beta stage, all the files under a particular directory will be read for the "to" links.

4.4 Special Cases

If the word contains a "/" in its first 2 letters, then we will not be able to create a folder of this type. Same is true with ".." or ".". So to be on the safer side, if the word contains a "/" or "." then it is treated as a symbols.

4.5 Assumptions

I am assuming that the directory structure of the google Ngram data remains same. This includes the file names and file counts. Also I assume that there is enough space to store the googlengram-reverse-network that I am creating. Also I am assuming that the user has write access on the directory from where he is running this program (because this is the place where the network will be created).

4.6 Efficiency Improvements

To improve efficiency of network building, I used the bulk insert algorithm while creating the network. In this case, instead of writing one entry to any to_Corientihan_[id] file, I write a bunch of entries (max 40) at a time. So all the consecutive entries which are to be written to the same file are first stored in a temporary string and then if this count reaches to 40 or if the file to be written to changes, this temporary string is flushed out to the file. This significantly reduces the file i/o in turn improving the efficiency.

4.7 Benchmarking

Program was run on the google-2gm data. The network was successfully generated for the whole 2gm data.

Processors	Timings (memory used: 2 gb)	Number of 2-gm files
16	3 hrs 56 mins	32
32	2 hrs 47 mins	32
64	1 hr 58 mins	32

As you can see the time does decrease with increase in number of processors. However, the time does not decrease by half when I double the number of processors. This is not because of the amount of data sent or because of the way I am distributing the data. This is solely because of the amount of disk access involved. The way I am building the network, there is a huge amount of disk access involved in each iteration. As the number of processors increase, more number of processors accesses the same part of disk making it really slow. Another important thing is that these times may vary depending on the amount of disk access (by all the users) at a given point time.

4.8 Space Requirements

As far as the memory is concerned, the networking building does not require any large amount of memory at all. I require about 100 mb of memory for my data storage. What the network building requires is a large amount of disk space. The amount of disk space required for building the reverse network for whole 2-gm data is around 5.5 gbs. This is because the reverse network actually has all the 2gm data itself, t is only stored in a completely different format. As a general rule, the reverse network will be approximately of the same size as the original data content.

4.9 Testing and Validation

I started out with smaller data sets and ran the program with various numbers of processors to check if I am getting the same output every time. I tried the cases where the number of processors are less than/ more than/ equal to the number of files. Then I ran the program on just one file in the google ngram data, with 1 and 8 processors. Got the same output. Then I ran the file on whole ngram data with 32 and 64 processors. To verify that the network built is correct, I calculated the number of nodes, edges and total edge weight produced by each processor. Then I wrote a stub that calculates all the number of lines which have tokens whose frequency is less than the unigram cutoff. This stub was a hard coded program that counted the number of lines. So the network built would have been wrong if these 2 numbers did not match. This was not true in my case. Also I did some adhoc testing by checking for several random bigrams. Another important factor to tell the correctness of network is the size. The way I am building the network, the size of the network that I build should be a bit more than the size of the bigram data (because data is scattered amongst several files instead of just 32). All these correctness tests were satisfied by the network that I built.

5.0 Network Interface [Salil Bapat]

The network interface contains functions to access the google-2gram network. The functions will access the network and return the required data.

5.1 Network Interface Details

The network may be accessed for 2 things. 1) To get all the links from a word. 2) To get all the links to a word. These 2 things are implemented in 2 separate functions named GetLinksFromANode and GetLinksToANode respectively. 2 separate functions were required because the way we store network for easy access of "to" and "from" words is completely different. All the "from" links are fetched from the already-existing google-ngram data which is provided. The "to" links are fetched from the google-ngram-reverse-network which is created in previous steps.

5.1.1 Getting the "from" links

The “from” links of any node are fetched from the google 2 gm data which is already provided. Following is the algorithm for this function:

1. Accept the source word as input of the function.
2. Open the 2gm.idx file which is inside the 2-gm data folder.
3. From the entries listed in this file, find the 2-gm file where this lookup word will start.
4. Open the file found in step 3.
5. Skip all the entries which do not match with the word to lookup.

6. Once you find the word to lookup, scan each next word and store all the connected nodes in a linked list.
7. If all the entries for the word to lookup end with the file still not having reached its end, return the head of the created linked list.
8. If the file reaches its end with the entries for word to look up still remaining, open the next file (files are sorted by numbers starting from 00 to 31) and adding entries in the linked list till the entries for word to look-up still exist.
9. Return the head of this created linked list.
10. Note that the first node in the linked list will always be the node to be looked up.
11. Each node in the linked list consists of 2 entries, the connected node and the frequency of the bigram. For the first node in the linked list, the frequency is set as -1.

5.1.2 Getting the "to" links

The “to” links to any node are fetched from the google-ngram-reverse-network which is created in earlier step.

Here is the algorithm for this function:

1. Accept the destination word as the input of the function.
2. Check the first 2 letters of the destination word. If the destination word is say "human" then the first 2 words will be hu.
3. Open the corresponding foder in the google-ngram-reverse-network. In above case you will open google-ngram-reverse-network/h/hu
4. Scan all the files which are created under this folder one file at a time. Number of these files will depend on the number of processors used to create the network.
5. All the entries in all the files will be scanned. When an entry is found with the destination node as the first token, then add that entry to the linked list. Repeat this process for all the files.
6. If you are unable to open the foder in question then that means there are no "to" links for this destination word. Hence the linked list in this case will consist of only one word which is the destination word.
7. If the word is a single letter word then the directory in question will just be google-ngramreverse-network/[word-in-question].
8. More on which foder will be accessed in which case can be found in the introduction comments of NetworkBuilder.c. Same algorithm is used over here.
9. Retrun the linked list which contains all the nodes to the destinations node.
10. Note that the first node in the linked list will always be the destination node which is being looked up.

11. Each node in the linked list consists of 2 entries, the connected node and the frequency of the bigram. For the first node in the linked list, the frequency is set as -1.

5.2 Data Structure used

A linked list is used for storing the "to" or "from" links of a node. The linked list consists of 2 items; the connected token and the frequency of the bigram. In every case, the first node of the linked list will be the token send in for a lookup. So even if there are no connections found, the linked list will still contain the head node with the token being looked up.

Example of this structure is as follows:

Suppose the bigrams are like:

good girl 100

good boy 200

good man 300

boy man 400

girl man 500

The "from" linked list for the word "good" is:

(good,-1)->(girl,100)->(boy,200)->(man,300)

The "to" linked list for the word "man" is:

(man,-1)->(good,300)->(boy,400)->(girl,500)

The data structure used for storing the network is ofcousre files and more on this data structure can be found in the introduction comments of NetworkBuilder.c

5.3 Network Usage

This network will be used in beta stage for dealing with the queries. While searching for all the paths of a given token, both these GetLinksFromANode and GetLinksToANode functions will be called to get all the connected links.

5.4 Testing and Validation

For testing the network interface, first a sample dataset was created. The network was tested on this sample dataset and logical defects were fixed right away. I then verified the network interface by creating a network of just one of the 2-gm files(2gm-0000). On this network I ran the network interface for various different words. I compared these results with results of a stubs that I wrote which was dedicated to calculate a connections of a particular word where in everything is hardcoded. The verification showed that there were some differences specially when the word has no connections at all which I later fixed. Then I went ahead and verified that this networked worked perfectly for the whole 2-gm data (with a approach similar to earlier).

6.0 Disjoint Networks

The part of problem in alpha stage is to find the number of disjoint networks and related statistics in the whole 2gm co-occurrence network. For this purpose, we divide unigram data amongst processors. Each processor forms the network for its own data. These networks are at the end combined together.

6.1 Data Distribution [Premchand Bellamkonda]

6.1.1 Unigram Cut-off

Before we implement the distribution of data, we first check for the unigram cutoff. Only the unigrams whose frequency are greater than the cutoff are distributed. If the frequency is nt greater than the cutoff, we just ignore the unigram and that unigram won't be distributed. To verify the condition, I take the cut-off form the command line and store it. Now in the unigram file we read every word and also read the frequency of that word beside it. I store these in the temporary buffer. If the condition is satisfied, I will store the word which is there in the buffer to the array which will be distributed by all the processors. If the condition is not satisfied, it will not store the word in the buffer and go to the next word in the unigram file and store that word and frequency in the temporary buffer.
In this way the unigram distribution is done using cutoff.

The code for the unigram cutoff is developed but it is not considered in the final version. This is because we are not able to change our architecture in the final stage because oof time complexity.

6.1.2 Associtive Cut-off

Associative Cutoff is done at the path finding module. Whenever a path is found, we can will find the asscoiation score. Asscociation score can be calculated by the number of times a bigram is occuring / Frequency of the individual scores. If the assoctaion score is greater than the cut-off, then we inlcude the path. Or else, it wont be included.

This part is not included because of the algorithm we developed in the initial stage. If we want to include this we will have to change the algorithm. As we did not get enough time to change the whole algorithm, we couldn’t implement the associative cut-off. We could have have done if had enough time to change the implementation (algorithm) of our project.

6.2 Distribution of Data by all Processors [Premchand Bellamkonda]

6.2.1 Approach

Distribution of the data to all the processors is done based on the number of processors and number of lines in the file. In our approach, we are first read the number of lines in the whole file and calculate the interval which is number of lines/No. of processors. We distribute the data in block-wise. Now master processor will give all the processors then stating point and the interval. For the last processor, it will give interval up to the end of the file. Now all the processors will be having their starting point and number of lines it has to read in the file.Now each processor will open the file and read till for its starting point till reaches its interval end. Here every processor will read the file and store in temporary

buffer till it reaches its starting point. Only from here, we are storing the values it read from the file till the end of its interval. As the file is having the number beside the word which is the number of times it occurred, we are also storing the number in a temporary buffer which we don't use again. Now each and every processor will have its words in the array. We are giving space after every word in the array for
passing as that's the specified format of input required.

.

6.2.2 Assumptions

I am assuming that the structure under the directory which user specifies at the command line remains same. This is because I am opening the unigram file located at /dvd1/data/1gms/vocab. Contents of this file will then be read and stored in the string as explained above.

6.2.3 Data Distribution design

Here we choose to distribute the data using block stream instead of cyclic distribution. This is because we are reading
the file in a sequential way and it will be easy to distribute it in block stream. Because if we use cyclic way, each

processor has to traverse through all the data until it gets the next item to be distribute. So in our case,
each processor don't have to traverse

6.1.4 Testing

The code for this part of the project is tested for different number of processors varying the size of the file. It is checked if the number of lines in the file is divided according with the number of processors and size of the file. It’s also checked that the last processor will indeed get all the remaining lines. We created different files of different size (with respect to number of lines). It works fine for all the cases.

Output: Gives an array of words to the Network_Stats function to find the disjoint networks.

6.3 Finding disjoint network per processor [Shruti Pandey]

6.3.1 Approach

After looking at the data we realized that if there are two unique words, say word1 and word2 which are connected to N other words forming networks say N1 and N2. Then provided any of the words in N1 is connected to any other word in N2 then that would mean both the networks are connected to each other. This was the basic approach we kept in mind.

Now simply connecting both the networks would have caused many duplicate entries of words in the formed network and looking at the size of the data it was very important to remove the redundancy so we took care in order to remove redundancy while processing the data.

Another issue was, what if there are two existing networks and later in some point of time we find a new network which is connected to both. It means that all three networks are connected so it would again be important to join all three networks removing the redundancy simultaneously in order to keep the processing time less.

One more important issue to address was the memory we allocate for the words to be stored in the link list. Since the largest word was of length 40 and the smallest was of just one character thus we decided to allocate the memory for a word dynamically as per the size of the word thus saving a lot of memory wastage. Performing malloc for each and every word is ofcourse an expensive operation as per time but since we had to utilize the memory efficiently we went on with this approach.

6.3.2 Module Description

This module finds the number of disjoint networks for each processor. It takes in the unigrams and for each unigram, it takes as input all the connected words to it and processes them in order to find the number of disjoint network.

6.3.3 Parallelism in the Program

In this module, each processor has its own set of data and finds the disjoint network for its dataset. After finding the number of disjoint Networks in each processor. The total number of processors are divided by 2 and in case the number of processors is an odd number then the ceiling for the above value is taken (Example: k = ceiling(n/2)) .All the processors which have the id more than 0 and less than k send write their respective network in a file which is named according to the processor number and the number of the network which is of the format (net<processor_id><network_no>) and then send the number of files created to processor 0 for further processing. Similarly the processors with id more than k do the same but send the number of files they generate to kth processor instead of 0. The network created by 0th processor and the kth processor is passed on to them as they were created for further processing. All these files are further agglomerated to get the total number of disjoint networks .

6.3.4 Assumptions

We assumed that the input provided to the program is correct and in the required format. Since we are dealing with link list in which you need to store the pointers too along with the data thus we assumed that as per the amount of data being processed by each processor the amount of memory allocated to the processor is enough for the processing of the program. Also I assume that there is enough space to store the files being generated by each processor . Also I am assuming that the user has write access on the directory from where he is running this program (because this is the place where the network will be created).

6.3.5 Input/Output

INPUT: Takes list of unigrams per processor. The input is in the format of tokens separated by spaces. example: "big blue cat cold dog hot yellow"

OUTPUT: The final output is a list of files containing the disjoint networks for all the processors other than 0 and kth processor where k = ceiling(p/2). 0th and kth processor just pass on their linked list to the program for agglomerating the data.

For e.g.: if there is only one processor: and the networks found in the processor is as follows:

```
Processor 0 : big-->dog-->cat-->runs (network 1)
              |
              blue-->jeans (network 2)
```

It writes these networks in 2 files named net01 and net02 with the contents as follows:

File net01: big dog cat runs

File net02: blue jeans

6.3.6 Data Structure Used

We have used a two dimensional linked list structure. The first linked lists (Network list) stores all the connected words. The second linked list (Base list) connects all the 1st linked lists. For each 1st link list formed a new node for the base list is formed which contains the head of the 1st linked list. Thus we can see that the number of nodes in the base list is actually the number of disjoint network in the processor.

For e.g. If the network list is like word1-->word2--> and another network list is word3-->word4. Then the base list will be

```
base1-->word1-->word2
|
base2-->word3-->word4
```

6.3.7 Algorithm

1) Get the set of unigram's and process one by one.

2) Till all the unigrams are not processed. Take one unigram from the list above say root token
and start processing following the steps given below.

3) For the first root token it searches the file (out of the 32 files) which contains the token by using the 2gm.idx and starts processing all the words connected to it one by one.

4) Till all the words connected to the root token are not processed. Take one word at a time and process as given below.

5) IF it is the first root token then I just create a base network and add the linked list
to it. This is done because I am considering only the links which are connected to the root
token (for links) so none of the words in the network would be redundant and thus we can form
a baselist in the starting.

6) ELSE Process the linked list till all the nodes for one root token is processed. Take one
node at a time and search in the already existing base list to check if it already exists in

any of the network.

a)IF WORD FOUND in the base list

->IF word found is the first word of the linked list or in other words is the root token then mark the network containing it. Find a pointer to the end of the network to make further processing fast. I added this step in the beta stage in order to fasten up the processing.

-> IF a network has already been marked and the word being searched is found in that network and there are notemporary networks made till now then break from the loop and go back to fetch the next word from the network as in step 4.

->IF word found is not the first word found and it is found to exist in a network other than the marked network then, check if there is any temporary network made. If there exists a temporary network then add this temporary network to the end of the marked list. Else if there is no existing temporary network then add the base list (in which the current word is found) to the already existing marked network and delete the base list pointer for the second base list.

b)IF WORD NOT FOUND in the base list.

-> IF word not found is the first word of the linked list or in other words is the root token then make a temporary network and set the network made flag to true.

->ELSE check how many networks are made in the base_list. If only one network has been made till now then check if temporary network is made already. If it is made then add the word to the already existing network and if not made then make a new network. In case there are more than one network then add the node to the marked network.

7) IF there are still more words connected to the root token then go back to step number 4 to

fetch the next word.

8) ELSE if none of the words from the currently processed list of words was found in the already

existing networks then add a new base list pointing to this network. Else if the number of networks in the base list is one and there are no temporary networks made then add the new

temporary network formed to the base list.

9) IF there are still root tokens to be processed then go back to step 2 to fetch more tokens.

10)Find k = ceiling(number_of_processors/2)

11) IF the id is greater than 0 and less than k then those processors would write each network

into a file named as described above and send the number of networks found to processor 0.

Similarly the processors whose id is greater than k also write their networks in the files as mentioned above and send the umber of networks found to processor k for further processing.

6.3.8 Testing

If we analyze the approach there were many cases which the program needed to handle like :

->before a token is found in a already existing network a temporary network has already been formed (the previous tokens were not found in any of the existing networks).

->all the tokens are found in the already existing base list.

->none of the tokens are found in the already existing base list.

And many more. To bring in the above cases as mentioned above I had to make may datasets to ensure the correctness of the program.

I am giving a set of inputs that I had used to test the correctness of the program and the output that gets formed. I had tested the program using one processor as well as more than one processor to check the correctness of the output files being generated by each processors. I worked through all these examples manually to ensure the output is correct.

Note: for simplicity I am giving the outputs generated as done by one processor to avoid confusion. There were many more test datasets which I had created but these would be enough to make the working and correctness clear.

Input from program “readnew.c” is same as we are using the same dataset for all the cases:

Input: "big blue cat cold dog hot yellow"

CASE 1: In this case you can see that as we work through the algorithm then by the time 4th batch of input is processed we have 3 networks and while processing the 5th batch input we can see that some of the nodes exist in network 1 and some in network two so we need to merge network one and two along with this batch of input and delete the base list pointer to the 2nd network.

The words connected to the root words given above from readNew.c:

1)big-->dog-->cat 2)blue-->jeans 3)cat-->runs 4)cold-->weather 5)dog-->barks-->runs-->bites-->blue 6)hot-->weather 7)yellow-->hat

Output:

```
base1-->big-->dog-->cat-->runs-->barks-->bites-->blue-->jeans
```

```
|
base2-->cold-->weather-->hot
|
base3-->yellow-->hat
```

Therefore, number of disjoint networks = 3.

CASE 2: This case is similar to above one but we can see that 4th ,6th and 7th batch of input are common for some words thus they are added into one. Thus we get 2 networks.

The words connected to the root words given above from readNew.c:

1)big-->dog-->cat 2)blue-->jeans 3)cat-->runs 4)cold-->weather 5)dog-->barks-->runs-->bites-->blue 6)hot-->weather 7)yellow-->hat-->cold

Output:

```
base1-->big-->dog-->cat-->runs-->barks-->bites-->blue-->jeans
|
base2-->cold-->weather-->hot-->yellow-->hat
```

Therefore, number of disjoint networks = 2.

CASE 3: This is the simplest example wherein all the inputs are connected and thus there is only one network formed.

The words connected to the root words given above from readNew.c:

1)big-->dog-->cat 2)blue-->jeans 3)cat-->runs 4)cold-->weather

5)dog-->barks-->runs-->bites-->blue 6)hot-->weather 7)yellow-->hat-->cold-->big

Output:

base1-->big-->dog-->cat-->runs-->barks-->bites-->blue-->jeans-->cold-->weather-->hot-->yellow-->hat

Therefore, number of disjoint networks = 1.

6.3.9 Observations

In the final stage we tested the program on a very large dataset. I tested it on the 2gm-0000 file and 2gm-0031 file
and successfully got the networks in these files. For the 2gm-0031 file I got 1869 networks. It took 17 hours for
the program to get the output and that is because I could use only one processor to execute the problem due
to the busy queue status.

According to our analysis the process is not very slow but the nature of the data decides the speed of the execution.
For example in the file 2gm-0000 there are 37240 unique tokens and these tokens are further connected to 100,00,000

tokens. Now if we analyse this data we would find that the minimum number of unique tokens is 37240 and apart from
that there might be many other unique tokens which would add up to the list and thus the linked list would grow
immensely thus slowing down the system.

On the other hand in the case of 2gm-0031 file it has 312504 root tokens which is more than the previous file
but in this case these are connected to 4843401 words and most of these words are disjoint and not connected
to each other. Thus forming a number of disjoint networks (1869) thus being comparatively faster.

Therefore, the more number of disjoint network we have, faster the program runs. or the more number of common words
we get,faster is the program.

we even executed the the system on 2gm-0000 file using 32 processors and the processing was lot faster than one
processor.

for eg: If we consider the 2gm-0031 file. There are 3,12,504 unique tokens and these tokens are connected
to 48,43,401 other tokens.
As mentioned above, the system took 17hrs to find the disjoint network for this data. If the number of processors

is 2 then the unique tokens would get divided into half which is equal to 156252 thus decreasing the number of
bigrams as the number of bigrams would be equal to the number of words connected to these unique tokens.
Thus in the above case it is pretty evident that as the number of processors increase the amount of data to
be processed keeps on decreasing thus the the network being formed would also get smaller and smaller leading to
greatly fastening up the process. Thus we can see that this method gets better and faster with more parallelism.

We will now see if this module follows the 2 basic laws of parallelism which are the Amdahl's Law and Gustafson-Barsis's:

If we look at Amdahl's Law which states that if f is the fraction of operations in a computation that must be performed
sequentially, where $0 <= f <= 1$. The maximum speedup S achievable by a parallel computer with p processors performing
the computaion is:

$$S <= 1/(f + (1-f)/p)$$

Thus looking at the above algorithm we can see that the parallelism is exploited in dividing the huge amount
of data. And the Serial component would be the network building. The main part taking time in the serial component
is the linked list searching time. But as we can see as the number of processors increase the searching time

(sequential part) decreases. Therefore we can see that the technique is in accordance with the Amdahl's law.

Coming back to Gustafson-Barsis's Law which states that given a parallel program of size n using p processors,
let s denote the fraction of total execution time spent in serial code. The maximum speedup S achievable by this
program is $S <= p + (1-p)s$

As mentioned above even in this case as the number of processors increase the sequential component decreases
and thus greatly increasing the speedup. Hence it is observed that even Gustafson's law is true for this module because if serial component is 0 then the speedup achievable is ideal but in this case the serial component cannot be 0 thus the achievable speedup is at most p.

6.3.10 Space Complexity

The space complexity of this program also includes the space complexity of “bigramsearch3.c” and “readnew.c” as they execute on the same processor. The total length mentioned below is the exact total length of all the unique words in the google-2-gram data found my calculating the length returned by the file “readnew.c” which returns an array of unique words separated by spaces.

```
Total num of unique words: 13588391
Total length       : 92728648
Avg word length    : 7 + 1 ('\0') = 8
```

base node:-

ptr: 8 bytes

ptr: 8 bytes

network node:-

word: 8 bytes

ptr : 8 bytes

network memory: (8+8)*13588391 (As the network has unique words) = 217,414,256

bytes

(218MB)

base memory (100 can change depending on the number of disjoint networks) :

(16+8)*100 = 2,400 bytes (2.4KB)

premchand's array: 92,728,648 bytes (93MB) CONTIGUOUS

BLOCK OF MEMORY on each processor

other sources of memory utilization:

file operations, miscellaneous variables, pointers,

function calls,etc.

So altogether in the worst case the total memory required should be around 700MB

6.3.11 Time complexity

P: “readnew.c”

n: words returned by readNew.c

m: words read from bigram data.

S: time complexity for “Find_Disjoint_Network”

p: num of processors and there is one base list

= (P + n * m * S) / p

= (P + nmS) / p

= O(n^2)

if p = 1 (1 processor), O (n^3)

if more than one base nodes.

b : base nodes

= (P + n* m * S *b) / p

= (P + nmSb) / p

= O(n^3)

if p = 1 (1 processor), O (n^4)

6.3.12 Benchmarking

Due to the busy queue status we could not get the benchmarking results but we tried it on one processor and it works properly. Only the problem being that it would take a lot of time depending on the amount of data due to one processor.

6.4 Combining the networks [Andrew Norgern]

6.4.1 Approach

This part of the program combines the efforts of all the other processors and determines the number of disjoint networks and the number of nodes in each network. Before this part of the program is run, it is assumed that each processor (besides processor 0 and processor k) has written its networks into individual files into a decided directory.

Processor 0 and processor k have their networks in a linked list structure.

Processor k is used to increase the parallelism by sharing the workload of processor 0 in combining the local disjoint networks of each processor. The value k is defined as the ceiling of (number of processors)/2.

Processors with id from 1 to k-1 will have sent its number of networks to processor 0.

Processors with id from k+1 to (number of processors)-1 will have sent its number of networks to processor k.

This program has the MPI code for recieving these numbers. When processor 0 and processor k recieve the number of networks from a processor, they will combine the networks of that processor to their list of networks. Each file that contains a disjoint network of that processor will be opened one by one to look and see if a node in that

file is in one of their networks. If it is, it combines the two networks, taking care of redundant nodes. If it isn't, it adds that network to the list of networks.
Then processor k will write it's networks to individual files. It then sends the number of networks to processor 0. Processor 0 will then open up those files and combine those networks with its own networks. After this, processor 0 will contain the list of disjoint networks. Processor 0 then prints out the number of disjoint networks and the number of nodes in each network.

6.4.2 Testing and Benchmarking

The overall system has been tested on a very small network, and everything works as expected. To test this individual part of the code on a large network, I designed a test case to use. For this part to work, there needs to be the local networks for each processor written to file. Also, the two processors that take care of combining the networks need to have their networks in linked lists. So I had to first provide these prerequisites and then allow the agglomeration to work. First, I used and modified Premchand's program to distribute the data of one of the 2 gram data files and write the output of each processors distributed data to files. This process was run as a separate program first. Then for the test case program, I had processor 0 and processor k , where k = ceiling of (number of processors) -1, open up their network files and store them in a linked list. Now all the prerequisites are met for testing my code. I first ran the test case with a wall time of 3 hours using 32 processors and it timed out. I then ran the test case with a wall time of 6 hours and it timed out. Finally I ran the test case with a wall time of 10 hours and it timed out. After this I ran out of time, and my processes were being stuck in the queue. I underestimated the amount of time it would take to agglomerate the networks in this way. I came to a conclusion about the design of our system that finds the disjoint networks. The part that distributes the data and finds the local networks needs more processors to work faster. But then the more processors that work to find the local networks, means the agglomeration of the networks will take longer. So there is a competing affect of using more processors for the two different parts of this module. There really isn't any benchmarking results for the agglomeration, since I was never able to run it to completion. If there was more time, I would have tried running it with just 4 processors. I would expect this to be able to run to completion. So the only numbers I have is that it takes longer than 10 hours for two processors to create a linked list of their network file and then combine all the networks to determine the disjoint networks.

6.5 Finding edges in the disjoint Networks [Premchand Bellamkonda]

For finding the number of edges in the disjoint network, I read the number which is the number of files which is generated after the aglommeration part.
I open all these files to find the number of disjoint newtorks. Each file will be having the disjoint network for which number of edges are to be found.

From that file, every word is considered and number of edges connected to that word are found.
A word is taken and number of “from” words linked to it are found. i.e Number of words connected to that particular words.
Example:

Dog -> barks

Dog -> shouts

Dog -> sleeps

Dog -> drinks

Dog -> crazy

So, here number of words connected or number of edges would be 5.

If at all we if any word is having 10 edges and another word is also having 10 edges but with commom words connected to it, that would
be taken care of because even if they have common words which are linked, they will come under different edges
Example:

Dog -> barks

Dog -> shouts

Dog -> sleeps

Dog -> drinks

Dog -> crazy

cat -> barks

cat -> shouts

cat -> sleeps

cat -> drinks

cat -> crazy

So here the total number of edges would be 10.

6.5.1 Procedure

We first found the number of lines in each file we read. Then we initialised an array and allocated the size of it dynamically with the number of lines in the file. Now for every word in the file, we send it to the function Get_Number_of_Edges() which will return the number of edges for that particular
word. What this function does is, we will pass a word to this function and this function will calculate the number of "from" links from that word. When we calculate all the from links of all the words in a network
then we will get the total number of edges in the disjoint network. I am not even considering the to links because this will cause false counting of edges. When the function Get_Number_of_Edges gets a word whose edges are to be found, the function first finds the file locaiton of this word in the network using the information file. Then this found file is scanned to find the first entry of the
word. On finding the first entry, we count all the entries where the word in question is the first token of the bigram. By doing this the total number of from edges of that word are found. This is then returned back and the counting of total number of edges of the network processes.

For finding the total number of edges in the file we sum up all the numbers got from all the previous words. This gives me the total number of edges in a file. Like this we read all the files and all the data present in the files and print the number of edges in each disjoint network.

6.5.2 Alternative approach

Another approach for finding the number of edges for the disjoint networks was to find them while building the disjoint networks itself. However in the approach which we are using, we deal only with single words and see its connections to other words. Depending on the connections, we add or remove a node from a network. If we keep track of number of edges during this time itself then this will increase the time for processing and the complexity of algorithm. Hence we decided to calculate the number of edges once we find all the different disjoint networks.

6.5.3 Testing

I tested this part of the code on the whole data set by using a few sample tokens and a few sample files which represent the disjoint networks. It is working properly and giving correct number of edges for small data set. But for the complete google data, it is crashing in between.

7.0 Utilities [Anagha Dharasurkar]

A co-occurrence network links together words that have occurred together in some piece of writing. Each word represents a node in the network, and the edges between the words indicate that they have occurred together. The edges are directed, and show the order in which the words have occurred. The weights on the edges are the frequency of the two words occurring together in that order. The words that occur together in a Google 2-gram are linked together, assuming that each of the word occurs individually 200 or more times.

The utilities file provides the functions needed by all the modules like network building and identification of disjoint networks. The google 2gram data is distributed in 32 files.
The file names start from 2gm-0000 to 2gm-0031. The data is stored in these files in sorted order. Along with these files, there is an information file, 2gm.idx, specifying the start token in each file. Thus the 32 files themselves act as a co-occurrence with the 2gm.idx acting like a mapping file for the data. The makePrimaryDirectories()function required by network building deals with forming the directory structure. The network will be stored in a directory structure with root directory as "google-ngram-reverse-network". This parent directory contains 63 pre-defined directories. 26 directories of alphabets "a" to "z", 26 of alphabets "A" to "Z", 10 of numbers 0 to 9. and an additional directory named "symbols". These 63 directories will be created right at the start of the program. After these are created, we start reading the bigram data. For each bigram found, another level of directories will be created. The function getnumberoflinesFileWise(parentDir,fileLines) returns the total number of lines file wise. The processor 0 counts the number of lines in all the files. From this count and from the number of processors, the number lines which each processor should process are calculated. The function GetGraphNodesByToken() is called to get all the related words for a specific token. This function accepts token and returns link list of all the tokens associated with the input token, according to the bigram files. The approach taken for this was as follows:-
Find the possible index file that contains this token. As the 2gm.idx file acts like a mapping file, by referring to the entries in this file will guide you to the actual file/files where this token is present.
You may only need 1 file but this function will support tokens spanning across upto 2 different bigram files. This is because a token may have its bigram entries started in one file and ended in next file. So it was important to take such cases into consideration. This function builds the linklist that contains all the occurrences of the given token. Here is the example of the same:

Suppose the token to look up is "big". First the appropriate file will be fetched. Then this file will be searched to find the token "big". Once this token is found, the linked list creation starts. The first node in the linked list will be "big" (token to be searched). The remaining nodes will be all the nodes which have a link from "big". So if there are words say "apple", "boy" and "girl" with link from "big" then the linked list will be: big->apple->boy->girl Apart from the approach I mentioned, I also evaluated an alternative approach where in I store the linked list at each stage in some temporary files. However that approach was not as efficient as this one and hence was discarded. Testing was done for all the functions taking into consideration the different calls from different modules to the functions.

8.0 Path Finding [Anagha Dharasurkar]

8.1 Introduction

A co-occurrence network links together words that have occurred together in some piece of writing. Each word represents a node in the network, and the edges between the words indicate that they have occurred together. The edges are directed, and show the order in which the words have occurred. The weights on the edges are the frequency of the two words occurring together in that order. The words that occur together in a Google 2-gram are linked together, assuming that each of the word occurs individually 200 or more times. The network which was constructed in the alpha stage was used in beta to support queries to the network. The queries 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 4 is requested, that should cause to display all the paths going into (and out of) the target word that have a total length of eight edges (where the target word is in the middle of two paths of length 4). If any complete paths are found that are less than the requested length, those are to displayed as well. Only path lengths of 1 or greater are allowed for display. This makes sense as noone is interested in seeing standalone nodes for display. A path length of 1 would simply indicate if the given target word occurs in the network or not. Only networks consisting of 1 or more words are allowed. Thus, a word that appears as a google unigram but is not involved in any ngram relationships would be considered its own network and should be be treated as a one node network with no edges associated with it. Also I implemented the search taking into consideration the case sensitive factor , that is "go" and "Go" were treated as different words. I made a target-list file (query_list file) that contained a list of target words and path lengths to be displayed. The format of target-list file was as follows:

target path-length

target1 path-length1

…

targetn path-lengthn

where target is the word that will be at the center of your network, path-length is the number of edges going into and out from the given target for which words will be displayed. For example, the entry girl 5 in target-list, would consider "girl" as the middle target node with nodes going into and out of the middle node and having path length 5 on each side that is the To and From nodes. So effectively I will get total length of 10 edges

eg, almost->every->nice->and->generous->girl->must->eat->good->italian->food

Also for final result along with the above format the frequencies between pair of words will be displayed with time it took to find the path. So it will look something like this, (below values are some arbitrary values for example purpose)

almost->(1000)every->(1800)nice->(2200)and->(1400)generous->(1900)girl-
>(2700)must-
>(3900)eat->(1200)good->(2100)italian->(4200)food

almost every will have frequency 1000,every nice will have frequency 1800, nice and will have frequency 2200, and so on etc.

8.2 Algorithm

The overall idea of my algorithm was as follows :-

1) Read the query (target-list) file according to the file format which is as follows <token> <path length>. This file is mapped from the command line argument provided.

2) Then distribute each query from target-list to processors parallely by block allocation scheme of n/p logic as follows,

```
int start = (rank - 1) * (totalQueriesRead/(totalProcesses -1));
int end = (rank) * (totalQueriesRead/(totalProcesses -1));
```

3) I have dedicated the processor with rank 0 exclusively for printing. So processors from rank 1 onwards will be involved in query processing. All processors send the path results they obtain to processor with rank 0 who is responsible for printing the individual paths explored.

4)I have used MPI_Send and MPI_Recv by which each processor passes the paths they obtained by bundling in a data structure (All data structures would be described in detail next in this document). I thought this brought about good deal of parallelism. The integrated nature of the problem and recursive nature of the algorithm made breaking the problem for more fine parallelism and division of work into pieces for allocation quite difficult without impacting the results.

5) The recursive traversal is done through all the 'from' nodes of the given target node. This functionality uses NetworkInterface helper to get list of all the 'from' nodes for a given targetnode. After this list is obtained it processes each 'from' list based on DFS technique. In addition to traversing the node, this function builds a tree for each node with edges that represents links and corrosponding frequencies.

6)Then the recursive traversal is done through all the 'to' nodes of the given target node. This functionality uses NetworkInterface helper to get list of all the 'to' nodes for a given targetnode. After this list is obtained it processes each 'to' list based on DFS technique. In addition to traversing the node, this function builds a tree for each node with edges that represents links and corrosponding frequencies.

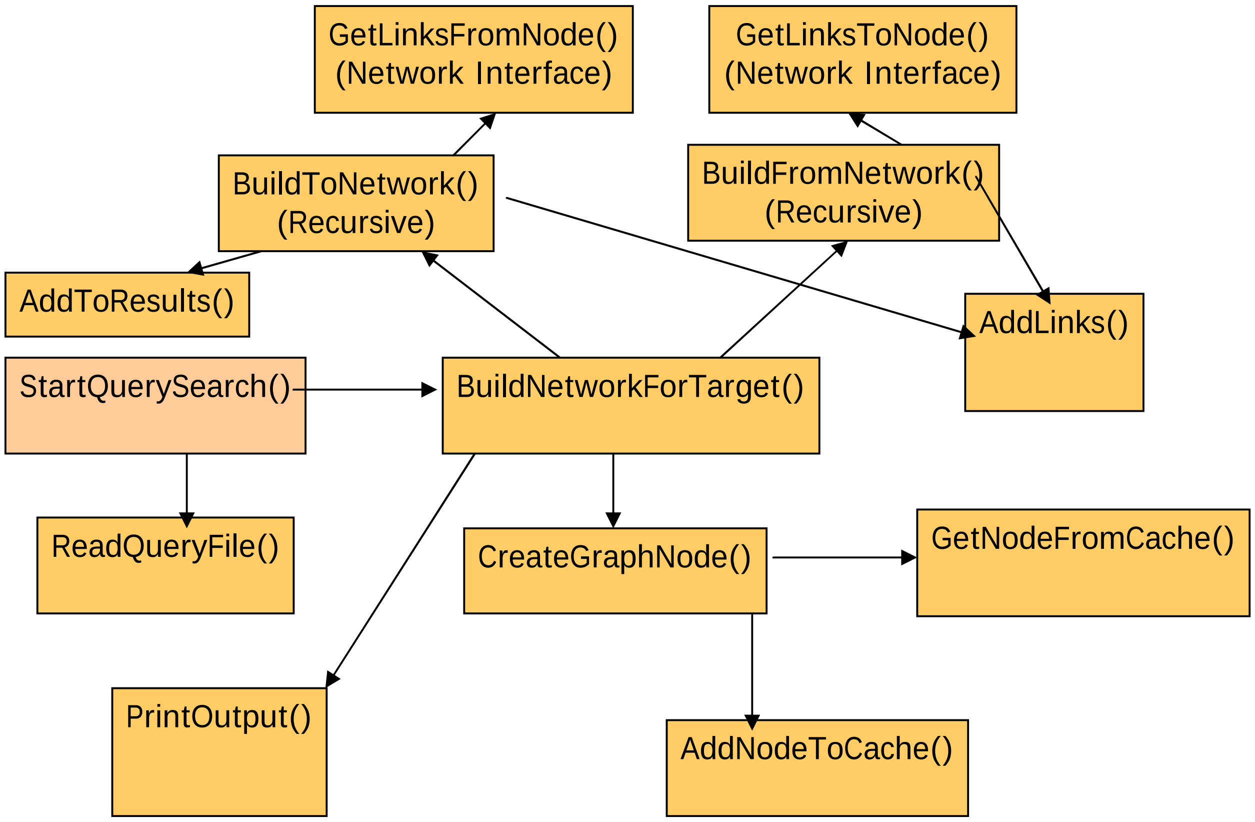
7) The AddLinks function creates necessary links between given 2 nodes. This function is called from both "BuildFromNetwork" & "BuildToNetwork" methods. This function basically joins the two nodes together as it can create both the "from" link & "to" links. It is called from both "BuildFromNetwork" & "BuildToNetwork" methods.

8) The recursion limit is till max path length mentioned. After that limit the above processing is stopped and the individual last nodes are stored in a data structure and now these will be the start nodes of different possible paths. This data stucture holding all the starting nodes of the individual paths in a link list of type "Results". (All Data structures described further).

9) For checking the cyclic dependency a node cache is maintained which stores the visited links. After the node is created it is also added to a cache. So the logic goes that if you create a link between nodes once and if there is request for linking them again then this is a potential cycle which will be confirmed along with few other checks. The GetNodeFromCache gets the requested node from cache instead of creating it again. (This could not be tested in detail due to lack of time)

8.3 Program Structure at a glance

(The below diagram just includes the important functions. The data structures used and input or output parameters are not mentioned in below diagram)



8.4 Data Structures

struct GraphNode (used for each node)

```
{
char *token;
int currentDepth;
struct GraphLink *fromEdges;
struct GraphLink *toEdges;
};
```

struct GraphLink (used for links between nodes)

```
{
```

```
struct LinkDetail *linkDetail;
struct GraphLink *Next;
};
struct LinkDetail (used by individual links)
{
struct GraphNode *neighbour;
int weight;
};
struct Results (used for final results)
{
struct GraphNode *start;
struct Results *Next;
int pathLength;
char *buf;
double elapsedTime;
};
struct NodeCache (used for cache implementation for cyclic dependency)
{
struct GraphNode *node;
struct NodeCache *Next;
};
```

8.5 Testing

Testing was done with different data sets ranging from small to very big data sets. I made some rough networks on paper and then made own dvd1 sets for them after constructing the bigram and idx files for my own networks. Then after running our alpha stage network creation the network was actually created and which in turn different target-list files were made for all combinations.

Then the answers for different paths explored with appropriate path lengths were matched with the various path traversal possibilities with the on paper network. The consideration of various data sets and their different query and path length combinations made sure that the code successfully passed all the input data variations without any bias for any specific conditions.

8.6 Challenges

The beta phase was challenging for me for implementation as it actually dealt with the path exploration and network traversal. Some issues in network creation for alpha stage had to be tackled well for accurate results in beta stage as now was the time when the created network was going to be used and verified. Some bugs were revealed for alpha stage network building during the beta testing. These were fixed during and beta code was tested again. So this helped strengthen the alpha network creation as well.

8.7 Memory Management Improvements

Beta code had plenty of memory leaks that caused issues when running path finding for larger density target words.

Current version of code will free resources as soon as path generation is now complete.

Following are details of improvements

AddLinks :-

This function is re-written to remove redundant 'link' code that was not necessary to build the network or find the path.

This function now allocates memory just enough to store the pointer to next element as oppose to adding link to backtrack the network.

Removing this code reduces the memory required to store links between 2 nodes from 32 bytes to 16 bytes.

BuildFromNetwork :-

This function is responsible for generating network for all the words that are to the right side of the target word.

This function is now freeing up memory occupied by the network interface return list as soon as that word is consumed

rather than consuming all the list and then freeing up the entire list.

This function had tremendous impact in processing more words because the amount of runtime memory available is

significantly more.

Another improvement done in this function is to release the memory occupied by the nodes on the path as soon as all the
possible paths for that node are generated and written to temporary output files.

This function also benefited from the optimization in AddLinks function (mentioned above).

BuildToNetwork :-

This function is responsible for generating network for all the words that are on the left hand side of the target node.
This function is now freeing up memory occupied by the network interface return list as soon as that word is consumed
rather than consuming all the list and then freeing up the entire list.
This function also benefited from the optimization in AddLinks function (mentioned above).

This function is similar to above mentioned function except it processes all the words on the left hand side of the target node.

8.8 Algorithm Changes

Path Finding Architecture Changes :

My Beta code strategy was to build the entire network in memory and then traverse through the network using a sort of

DFS (depth first search) method to generate. This function when tested on the actual google data fell apart because of
huge memory requirement to build the entire network in memory. For this release I have optimized my code to generate network
in 2 different halves,

1 - Left side of the target word &

2 - Right side of the target word.

How does it work better?

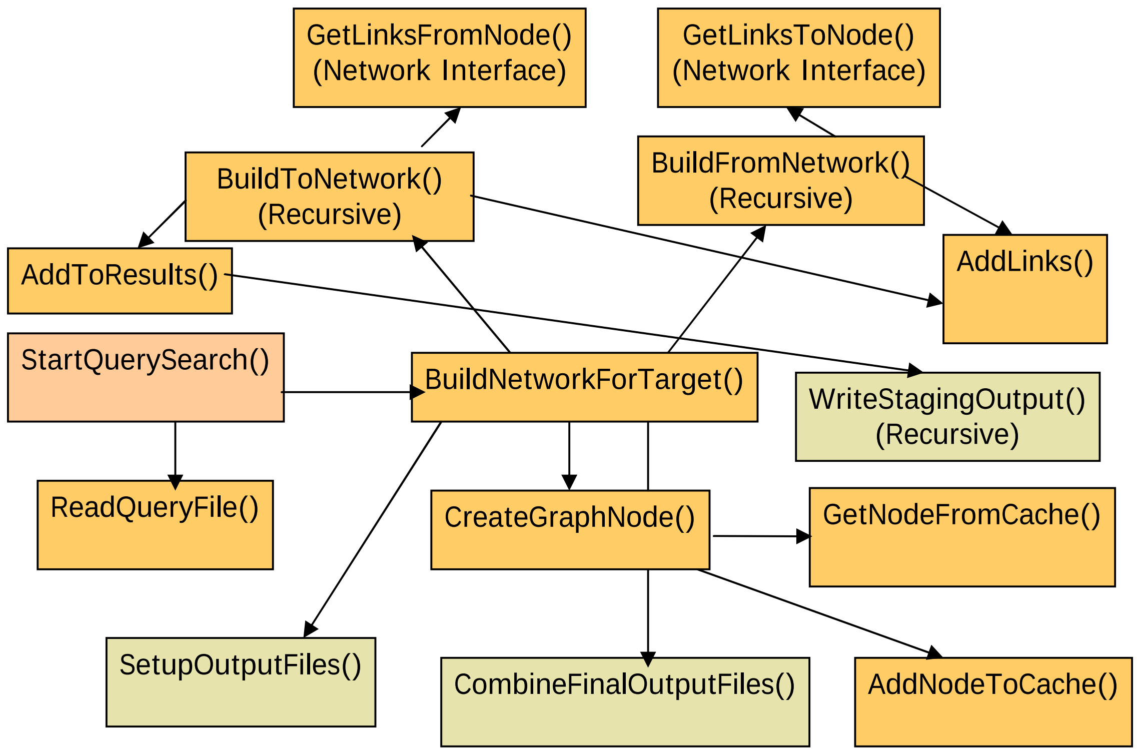
My beta code supported processing left and right network separately with few dependencies between them.
In this version of the code, the 2 processing functions are completely separate and have no dependency.
Left side processing function (BuildToNetwork) processes data and writes it to an intermediate file on the
disk and releases memory once path is found. Right side processing function (BuildFromNetwork) processes data and also
writes it to and intermediate file and releases memory.
These intermediate files are named as "<target_word>_left" and "<target_word>_right" respectively.
Once both the functions are done writing intermediate paths to the files, the last function combines the paths from
left and right files to produce final output (on console).
This made possible producing definite results for path length greater than 1.

Memory Optimization :

For the beta code, in order to generate one path the code had to processes the entire network from left most word to the right most word.
Assuming a network that has path length of 2 with average number 'from' or 'to' words = 10 beta code would had to processes 221 nodes to build the
entire graph before generating even a single path. But in the new architecture at any time we only need 110 elements to
form a complete path from the left side or the right side before the partial path is generated and written out to temporary file.
This allows code to quickly release memory.

8.9 Revised Block Diagram For Changed Algorithm

(The below diagram just includes the important functions. The data structures used and input or output parameters are not mentioned in below diagram)


GetLinksFromNode() (Network Interface)
GetLinksToNode() (Network Interface)
BuildToNetwork() (Recursive)
BuildFromNetwork() (Recursive)
AddToResults()
AddLinks()
StartQuerySearch()
BuildNetworkForTarget()
WriteStagingOutput() (Recursive)
ReadQueryFile()
CreateGraphNode()
GetNodeFromCache()
SetupOutputFiles()
CombineFinalOutputFiles()
AddNodeToCache()

8.10 Optimization for Parallelism

As the network is processed in 2 parts it is theoretically possible to process the 'from' & 'to' functions (BuildFromNetwork, BuildToNetwork).

These functions are independently executed on different processors to find left and right side networks of the target paths.

This functionality required MPI scheduling and data gathering routines.

Processing these networks separately on different processors also allows more memory to process
larger network because 8GB on the blade will be allocated to build only network that is on one side of the target word
as opposed to entire network.

8.11 Testing

A)Unit Testing

Each function in this code was tested with predetermined output for given set of parameters.
I tested this code against 2 sets of network, one that was hand designed (manually) and then mapped it onto exactly
google n-gram data format. These networks were quite small and served the purpose of making sure that each function is outputting data correctly.

I also wrote awk scripts and regular expression parsing to verify the actual words scanned from the google data files to compare it
to the actual processing in the code.

```
-------------------------------------- AWK SCRIPT -------------------------------------------------
--------
awk '
{
```

```
for (i = 1; i <= 1; i++)
freq[$i]++
}
END {
for (word in freq)
printf "mkdir %s\nmkdir %s\\to \nmkdir %s\\from\n", word, word, word
}' vocab_cs > create_dir_script
```


B) Integration Testing

I am using interface function to load data from underlying networks to process data, that is written by one of the team member.
 These functions did not change with the exception of few minor bug fixes. This interface stayed constant did not require separate testing as unit testing routine covered these function calls also.

C) Stress Testing

I have extensively tested the code to find out the boundaries or maximum number of words that could be processed before the memory was

exhausted on the given machine. This particular testing method had variable results and was dependent on other external factors outside
of my controls such as, number of other jobs running on the node at the time of execution of code, amount of memory available for allocation
and memory/load on the system. Since I am generating intermediate files I gathered all the data from the intermediate files and following is one of the output that crashed after processing average number of nodes

I ran 6 different tests with target word = www.nokia.com which consistently processed more than 4,000,000 words before it died.
All the nodes below www.nokia.com, such as from, of, on, etc are the associated or children nodes.

Function Name	Lookup Word	# of Nodes Returned
BuildToNetwork	www.nokia.com, length is	28
BuildToNetwork	from, length is	267470
BuildToNetwork	of, length is	469588
BuildToNetwork	on, length is	435238
BuildToNetwork	or, length is	354156
BuildToNetwork	nokia.com, length is	24
BuildToNetwork		1363433
BuildToNetwork	[, length is	470332
BuildToNetwork	eg, length is	8613
BuildToNetwork	see, length is	24484
BuildToNetwork	site, length is 54148	
BuildToNetwork	info, length is	23943

```
BuildToNetwork          through, length is           77796
BuildToNetwork          to, length is                603945
BuildToNetwork          Visit, length is             12253
BuildToNetwork          visit, length is             13636
BuildToNetwork          www.n-gage.com/press, length is  4
BuildToNetwork          www.nokia.com, length is     28
BuildToNetwork          Check, length is             16369
BuildToNetwork          as, length is                268761
______________________________________________________________________
                    Total Words Processed    4,464,249
```

The above result shows that the 'To' Nodes processing itself consumed more than 4 million before it crashed. Hence the
BuildFromNetwork did not get a chance at all.

8.12 Migration to Altix

Since amount of memory available on ALTIX is lot more than blade, massive data could be managed to find paths for greater than 1 path length.
The output path files were as big as upto 30 GB. Some of the interesting results for some the target words were :-

TARGET WORD	PATH-LENGTH	OUTPUT FILE SIZE (APPROX)
aaryan	3	24 GB

macstate	2	22 GB
www.nokia.com	3	25 GB
Bush	2	18 GB
india	2	12 GB

On Altix I was able to get results for even small density keywords upto 8 pathlength before it ran out of space.
As it can be seen it was not possible to provide all these results on Blade. But migration of our entire system on Altix was not chosen because it was late by the time I got the results. Also altix has queue etc response problems.

8.13 Benchmarking Results

Program was run on the real google-2gm data. The path finding was successful and all the paths were generated for the following target list
with 2,3,4 and 5 processors and various execution 'TIME' were obtained. These results below are on Blade.

-----Target list-------------------

```
gulti 2

zakas 2

tagor 2

pendse 2

-----------------------------------------------
```

The target.txt file was kept consistent to find the performance change with respect to increasing number of processors.

```
        Processors     Total Execution Time (to find all the paths)
----------------------------------------------------------------------

          2                    Total Walltime = 6.992774
          3                        Total Walltime = 4.617482

          4                        Total Walltime = 4.246133

          5                        Total Walltime = 3.163215
```

As you can see the time does decrease with increase in number of processors. However, the time does not decrease by half when
I double the number of processors. Along with some communication overhead this is mainly because of some inherent
sequential portion of the code which was not possible to parallelize. Later down kindly find the Performance Analysis done
by Amdahl's Law and Gustafson-Barsis's Law.

****

9) PERFORMANCE ANALYSIS

****

It was not possible to make some inherent sequential portion of the code parallel. This component was mainly recursion for getting 'from'
and 'to' nodes recursively, associated with the target word till the specified path length. It is difficult or can say almost impossible
to divide recursion into parallel task pieces. This chunk was quite considerable and had quite some impact. Also some small time is spent

performing initializations and cleanup on processor. So overall on the average can say 60% code is sequential.

Hence the fraction f = 0.6 (portion of the computation that is inherently sequential). This fraction will help us do the performance analysis by

Amdahl's Law and Gustafson-Barsis's Law for varying number of processors.

$$\text{SpeedUp (in general)} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

Amdahl's Law :-

Amdahl's Law is based on the assumption that we are trying to solve a problem of fixed size as quickly as possible.

It provides an upper bound on the speedup achievable by applying a certain number of processors to solve a

problem in parallel.

Let f be the fraction of operations in a computation that must be performed sequentially, where 0 <= f <= 1. The maximum speedup S achievable by a parallel computer with p processors performing the computation is,

$$S = \frac{1}{f + (1-f)/p}$$

Using this above formula , after calculations the speedup that can be acheived by Amdahl's Law for different number of processors is,

Processors	Speedup (S)
2	1.25
4	1.428
6	1.5
8	1.54
16	1.6

It can be seen that the speedup values are not differing greatly and hence can say that speedup achieved has not benefited by increasing number of processors due to the considerable sequential portion of the code which is a hurdle.

Gustafson-Barsis's Law :-

While Amdahl's Law determines speedup by taking a serial computation and predicting how quickly that computation could execute
on multiple processors, Gustafson-Barsis's Law does just the opposite. It begins with a parallel computation and estimates
how much faster the parallel computation is than the same computation executing on a single processor.

Given a parallel program solving a problem of size n using p processors, let s denote the fraction of total execution time spent
in serial code.
The maximum speedup S achievable by this program is

$$S <= p + (1-p)s$$

We refer to the speedup predicted by Gustafson-Barsis's Law as scaled speedup, because by using the parallel computation as
the starting point, rather than the sequential computation it allows the problem size to be an increasing function of the number of
processors. The fraction s = 0.6 (derived above) for inherent sequential portion of the code remains the same.

Using the above formula , after calculations the scaled speedup that can be acheived by Gustafson-Barsis's Law for different
number of processors is,

Processors	Scaled Speedup (S)
2	1.4
4	2.2
6	3.0
8	3.8
16	7.0

It can be seen that the scaled speedup values are nicely improving by increasing number of processors. This
is probably because Gustafson-Barsis's Law uses the parallel computation as the starting point rather than the sequential.

Here also the considerable sequential portion of the code is a barrier but its effect on speedup is less worse than in
Amdahl's Law .

9.0 Running the Project [Salil Bapat]

The tar file containing all the project code and documentation for the beta stage will be named google-ngram-1.0.tar. When this file is untared, it will contain 2 directories Code and Report. The Code directory will contain all the project code. In the same directory there will be a runit.sh shell script. Running this script will run this project.

For running the runit.sh, you need to give it following command line parameters:

a) dir-name: The name of the directory which contains the unpacked dvd directories that contain the Google Ngram data. It is assumed that the dvd directories and all of their subdirectories will remain exactly as provided by google. The project will simply read this data and not modify it in any way.

b) unigram cutoff (integer/number): unigram-cut is the minimum number of times each unigram in a bigram must have occurred for a bigram to be included in the network you build.

c) associative cutof (number): it is the minimum "association score" that a pair of words must have in order to be included in a path.

d) target-list: target-list is a file that will contain a list of target words and path lengths to be displayed.

So the command will look like:

runit.sh dir-name unigram-cut assoc-cut target-list

On running this command, a directory named google-ngram-reverse-network will be created which will contain the reverse google-ngram network. After this the disjoint networks and network statistics will be found out. Then the queries in target-list will be processed.

10.0 Bibliography

[1] Official Google Research Blog. 03 August, 2006. Brants,Thorsten; Franz, Alex. 14 December 2007.
<http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html>

[2] Denhiére, Guy; Lemaire, Benoît. (2004). “Incremental construction of an associative network from a corpus.” 26th Annual Meeting of the Cognitive Science Society, CogSci2004. Hillsdale, NJ. Lawrence Erlbaum Publisher.
<http://cogprints.org/3779/1/cogsci04_2.pdf>

[3] Edmonds, Philip. (1997). “Choosing the word most typical in context using a lexical co-occurrence Network.” Annual Meeting of the ACL, Madrid, Spain: Pages 507-509. Association for Computational Linguistics.
<http://portal.acm.org/citation.cfm?id=976909.979684>

[4] Ferret Olivier. (2004). “Discovering word senses from a network of lexical cooccurrences.” International Conference On Computational Linguistics, Geneva, Switzerland: Article Number 1326. Association for Computational Linguistics.
<http://portal.acm.org/citation.cfm?id=1220549>

[5] Nitta Yoshihiko; Niwa Yoshiki. (1994). “Co-occurrence vectors from corpora vs. distance vectors from dictionaries.” International Conference On Computational Linguistics, Kyoto, Japan: Pages 304–309. Association for Computational Linguistics.
<http://portal.acm.org/citation.cfm?id=991938>

[6] van der Weerd, Peter; Veling, Anne. (1999). “Conceptual Grouping in Word Co-Occurrence Networks.” Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, Stockholm, Swededn: Pages 4–9. Morgan Kaufmann Publishers Inc. <http://portal.acm.org/citation.cfm?id=646307.687429>