All Pairs Shortest Path Algorithms

5th November 1999 16:01

Professor Tadao Takaoka and David Cook

Abstract

There are many algorithms for the all pairs shortest path problem, depending on variations of the problem. The simplest version takes only the size of vertex set as a parameter. As additional parameters, other problems specify the number of edges and/or the maximum value of edge costs.

In this report, we focus on the edge costs. Specifically, we identify the distribution of edge costs. If the spectrum of edge costs distributes around two values, we can speed up the processing time. This is typical in road networks, where we have big distances between main centres, and small distances within the centre cities.

Contents

1 Introduction **3**

2 All Pairs Shortest Path Algorithm **4**
2.1 Work Previously Done 4
2.2 All Pairs Shortest Path Problem 4
2.3 Distance Matrix Multiplication 5
2.4 Basic All Pairs Shortest Path Algorithms 6

3 Implementation of 3 All Pairs Shortest Path Algorithms **8**
3.1 Alon-Gali-Margalit Algorithm 8
3.1.1 Results 9
3.2 Non-negative Real Numbers 9
3.3 Algorithm for Graphs with Small Edge Costs 10
3.3.1 Decrease in Time Complexity 10

4 Spectral Graph Theory **13**
4.1 What Is Spectral Theory 13
4.2 New Algorithm - Spectral Analysis of Matrices 13
4.3 A Problem With This New Algorithm 14

5 Two-band Algorithm **17**
5.1 Requirement For the Sub-graph 17
5.2 The Two-band Algorithm 17
5.3 Deciding How To Split The Subgraphs 18
5.4 Predictable Gain 18

6 Maximum Sub-Array problem - An application of The All Pairs Shortest Path **20**
6.1 Maximum Sub Array Algorithm 20
6.2 Results 21

7 A Graphical Front End For Displaying Graph Algorithms **23**
7.1 Future Development 23

8 Conclusion **25**
8.1 Further Work 25

A Code listings **27**

Chapter 1

Introduction

This report presents the reader with a new approach to solving the all pairs shortest path problem. This approach involves looking at the structure of the graph and then using this information to help decrease the time complexity of the problem.

As discussed in section 2.1 a number of approaches have already been taken that take into consideration the structure of the graph, yet none of these have ever looked at the values that make up the distance matrix.

The structure of this report is as follows, in the second chapter a brief introduction to what the all pairs shortest path problem is given, along with its relationship to distance matrix multiplication. The relationship between distance matrix multiplication and arithmetic matrix multiplication is also discussed in this chapter.

In the third chapter three algorithms are discussed that the author has implemented during the year. A small decrease in the time complexity of the third algorithm was found.

Chapter four expands on the work done in the previous chapter. It develops the idea of spectral theory and then further refines this into an algorithm for performing distance matrix multiplication. At the end of this chapter a problem with the spectral theory algorithm is discussed that prevents it from being used to solve the all pairs shortest path problem.

In chapter five another algorithm is presented that modifies the algorithm from chapter four. This new algorithm is then able to calculate the all pairs shortest path problem for graphs that have a specific structure.

Chapter six of this report offers the reader an example of an application area where the all pairs shortest path problem could be used.. The maximum subarray problem was recently found to be equivalent to the all pairs shortest path problem and as such could benefit from any decrease in the time complexity of the all pairs shortest path algorithms.

Chapter seven presents something completely different from what the other chapters were presenting. In this chapter a brief look at a graphical user interface tool for displaying running all pairs shortest path algorithms is presented.

Finally chapter eight presents some brief conclusions about what has been achieved in the project as well as a look at some future work that could be done in this field.

Chapter 2

All Pairs Shortest Path Algorithm

2.1 Work Previously Done

Over the years there has been a large amount of work done in the field of the all pairs shortest path problem. This work has seen people conclude that the all pairs shortest path is the same as distance matrix multiplication[1]. Authors have discovered new upper bounds on the time complexity of the problem [2]. Algorithms that deal with nearly acyclic directed graphs have been found [3]. Finally algorithms with expected running time of $O(n^2 log\, n)$ have been found[4]. No work has been done on actually looking at the values that make up the distance matrix.

2.2 All Pairs Shortest Path Problem

The all pairs shortest path problem involves finding the shortest path from each node in the graph to every other node in the graph. A directed graph is presented in Figure 2.1. This can be calculated by using a number of different algorithms. One way is simply to perform a single source algorithm, this problem calculates the shortest path from one node to every other node, and applies it to all the nodes in the graph.

Another way is to present the graph as a matrix, this can be seen in Figure 2.1 which shows a graph. The corresponding matrix representation of the graph is given in Figure 2.2. Once this matrix has been constructed, distance matrix multiplication can be performed on it, distance matrix multiplication is explained in section 2.3. The method of repeated squaring is used on the matrix $\log n$ times. Once this repeated squaring has been completed the matrix will then contain the solution to the all pairs shortest path problem.

There are other methods for calculating the all pairs shortest path problem, two of which are presented in section 2.4.

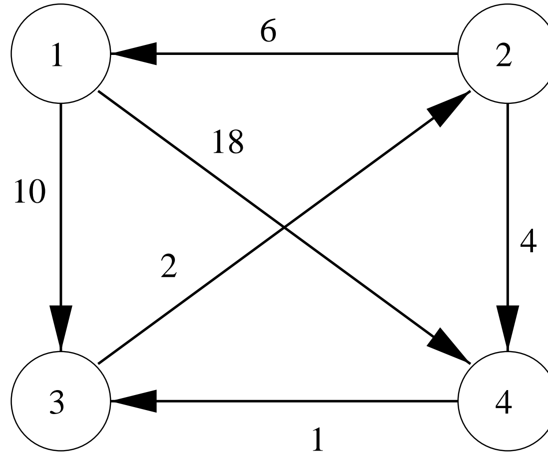

Figure 2.1: A directed graph

0	-	10	18
6	0	-	4
-	2	0	-
-	-	1	0

Figure 2.2: The matrix for the graph from figure 2.1

2.3 Distance Matrix Multiplication

Distance matrix multiplication is similar to matrix multiplication except where multiplication is used addition is used and where addition is used the *min* operator is used. This operator selects the minimum number from a set of numbers. The above information is presented in the table shown in Figure 2.3. Also presented in the table is boolean matrix multiplication.

Distance matrix multiplication has the same time complexity as matrix multiplication, and as such algorithms for both of them can easily be adapted to perform either matrix multiplication or distance matrix multiplication. The simplest algorithm for performing matrix multiplication is set out in Figure 2.4. This algorithm must perform line 4 n^3 times and as such has time complexity of $O(n^3)$.

This simple algorithm for matrix multiplication can easily be shown to have $O(n^3)$ time complexity. This simple matrix multiplication algorithm can easily

	*	+
Distance	+	MIN
Arithmetic	*	+
Boolean	AND	OR

Figure 2.3: Table displaying three types of matrix multiplication

```
0 Initialise Matrix;
1 for(i=0;i<n;i++) {
2  for(j=0;j<n;j++) {
3    for(l=0;l<n;l++) {
4      Matrix[i][j] += Matrix[i][l] * Matrix[l][j];
```

Figure 2.4: Simple program code to perform matrix multiplication

```
0 Initialise Matrix;
1 for(i=0;i<n;i++)
2  for(j=0;j<n;j++)
3    for(k=0;l<n;l++)
4      if(Matrix[i][k] > Matrix[i][j] + Matrix[j][i])
5        Matrix[i][k] = Matrix[i][j] + Matrix[j][i];
```

Figure 2.5: Floyd's all pairs shortest path algorithm

be adapted to the distance matrix multiplication algorithm.

2.4 Basic All Pairs Shortest Path Algorithms

There exists some basic algorithms for solving the all pairs shortest path problem. Two examples of these are Floyd's[5] see Figure 2.5, and repeated squaring of the distance matrix see Figure 2.7. These algorithms have time complexity of $O(n^3)$ and $O(n^3)$ $log\ n$. The basic code for both of these algorithms is set out in the two figures.

By using Floyd's algorithm, presented in Figure 2.5, on the graph from Figure 2.1 a solution matrix is given in Figure 2.6.

0	12	10	18
6	0	5	4
8	2	0	6
9	3	1	0

Figure 2.6: The solution matrix for the graph from Figure 2.1

```
0 Initialise Matrix;
1 for(k=0;k<(log(n));k++)
2  for(i=0;i<n;i++)
3   for(j=0;j<n;j++)
4    for(l=0;l<n;l++)
5      if(Matrix[i][j] > Matrix[i][l] + Matrix[l][j])
         Matrix[i][j] = Matrix[i][l] + Matrix[l][j];
```

Figure 2.7: Repeated squaring of the distance matrix

Chapter 3

Implementation of 3 All Pairs Shortest Path Algorithms

Presented in this chapter are three algorithms that at the start of year the author was looking at. From the third of these algorithm's, the new algorithms presented in chapter four and chapter five were found. These algorithms can be found in a paper presented by T. Takaoka[6].

These algorithms are based on to distinct phases an accelerating phase and a cruising phase. Both these phases are combined to solve the problem. The accelerating phase uses a previously written algorithm and stops this algorithm at a particular point called r. Then it switches to the cruising phase, which is used to finish calculating the solution. This is summarised in Figure 3.1.

3.1 Alon-Gali-Margalit Algorithm

This algorithm consists of two parts, the first part of the algorithm is the accelerating phase, which is based on the algorithm by Alon-Galil-Margalit[7]. This algorithm can be used to solve the all pairs shortest distance problem. By using a second phase to this algorithm a decrease in the amount of time taken to compute the all pairs shortest distance can be achieved. This second phase is refered to as the cruising phase. The time complexity achieved by this algorithm is $O(n^{(\omega+3)/2}\sqrt{(log\ n)^c})$. The meaning of ω is the current power of n

```
1 for i = 1 to r do
2   Accelerating(i);
3 End;
4 for i = r to log n do
5   CruisingPhase(i);
6 End;
```

Figure 3.1: The accelerating and cruising phase modal

for the time complexity to perform a distance matrix multiplication. The value for this currently is $\omega = 2.376$ based on an algorithm presented in a paper by D. Coppersmith and S. Winograd [8]. For the implementation of the algorithm that the author wrote, Strassen's algorithm for matrix multiplication was used which has time complexity of $0(n^{2.81})$ [9]. This actual implementation has been omitted from this report and can be found in the paper by T. Takaoka[6].

3.1.1 Results

This Alon-Gali-Margalit algorithm was run against the Floyd's algorithm presented above, see Figure 2.5. Strassen's distance matrix multiplication algorithm of time complexity $O(n^{0.83})$ was used to perform the matrix multiplication. The graph clearly shows the AGM algorithm is better than Floyd.

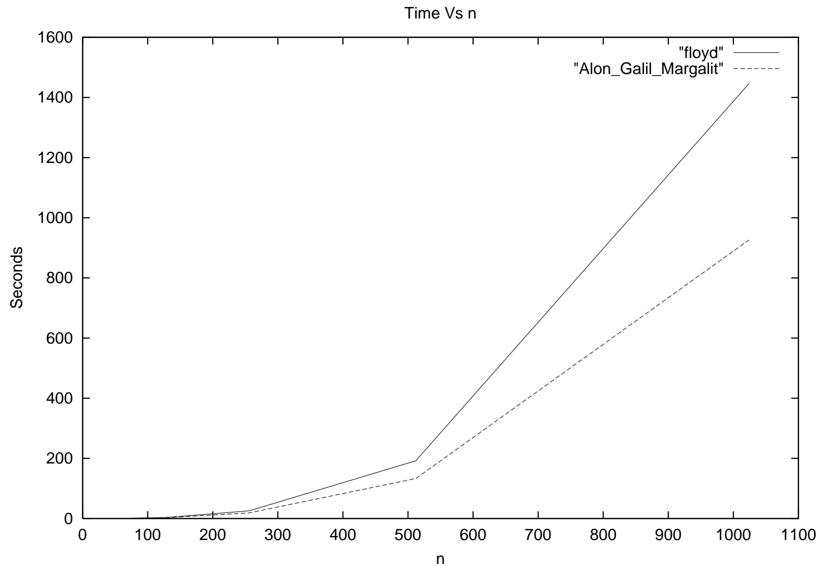

Figure 3.2: Results from comparing the two algorithms

3.2 Non-negative Real Numbers

This second algorithm is again similar to the first in that it is split into two parts the accelerating phase and the cruising phase. This algorithm can however unlike the previous one which only deals with integer edge costs, handle edge costs of any value. This algorithm along with the first is able to be run in parallel. Like the first algorithm the implementation of this algorithm has been omitted as most of the work presented in this report stems from the 3rd algorithm from the paper by T. Takaoka[6].

3.3 Algorithm for Graphs with Small Edge Costs

The third algorithm uses the same cruising phase as the previous algorithm, but uses a different accelerating phase. The accelerating phase for this algorithm only works if the edge cost is small with this algorithm the edge cost must be $n^{0.624}$ where n is the number of nodes in the graph. The largest edge cost must be kept at this value in order to keep the overall algorithm at sub-cubic time complexity. The number of nodes in the graph is dependent on the size of the edge costs. As the edge costs increase the algorithm becomes slower. This dependency is due to the fact that Schonhage and Strassen's FFT algorithm is used[10]. This algorithm is able to solve the problem of multiplication on a k bit number in $O(k\ log\ k\ log\ log\ k)$ time. This is quite an improvement over standard multiplication which has a time complexity of $O(k^2)$. However this improvement in time complexity is only noticed when the number of bits to be multiplied is greater than 1000. Timing these programs against other all pairs shortest path algorithms will not show any decrease in time complexity until numbers consisting of 1000 bits or more are used. The time complexity of this algorithm is $O(M^{\frac{1}{3}} n^{\frac{6+\omega}{3}} (log\ n)^{\frac{2}{3}} (log\ log\ n)^{\frac{1}{3}})$. This algorithm is dependent on M, the largest value in the distance matrix.

3.3.1 Decrease in Time Complexity

As discussed in the previous subsection any decrease in the size of the edge costs in the matrix will decrease the time complexity and hence the size of the maximum edge cost which at present is at $n^{0.624}$. A small increase in efficiency for this algorithm has been discovered. If the edge costs fall between $n^{0.624}$ and some other integer less than this but greater than zero, then by subtracting off this smaller value you are left with a smaller number, which as discussed earlier provides for some increase in efficiency, due to the fact you may now need less bits to store that number.

An example of this in effect is as follows. If you look at the matrix in Figure 3.3, the minimum value in this matrix is 6, as the zeros in the matrix represent an edge from the node back to the same node they are not taken into consideration. The solution to the all pairs shortest path problem for the matrix is given in Figure 3.4.

The maximum value in the matrix is 18. 18 requires 5 bits to be represented as the binary number 10110. After subtracting off 6 from all the values in the matrix you are left with a new matrix shown in Figure 3.5. The largest value in this new matrix is now 12 which requires only 4 bits to be represented as the binary number 1100.

The calculation of the algorithm can be performed on the reduced matrix in Figure 3.5. This reduced matrix consists of the matrix and the offset matrix. The offset matrix contains the number subtracted off of the original matrix. The matrix after completing one matrix multiplication is shown in Figure 3.6. After further repeated squaring the same matrix as Figure 3.6 is found.

Once this addition has been performed, each element in the solution matrix has the *min* operator performed on it with its corresponding value from the original matrix as shown in Figure 3.7. The final solution after performing the *min* operator is given in Figure 3.8. This is the same matrix as that given in Figure 3.4.

It is trivial to show that the extra three steps of performing the subtracting, the addition and the *min* operator all take $O(n^2)$ time, so they can be absorbed into the dominant part of the algorithm which is the matrix multiplication.

Very little increase in efficiency is obtained from this algorithm because in many cases there may be a large gap between the smallest and largest numbers in the graph. This algorithm is dependent on the length of the number range, if the range is small, some decrease in time complexity could be observed and if the range is large, then little decrease in time complexity would be observed. However this did lead us to discover Spectral graph theory which is presented in the next chapter.

0	6	16	8
9	0	7	18
10	7	0	18
17	7	13	0

Figure 3.3: The initial matrix

0	6	13	8
9	0	7	17
10	7	0	18
16	7	13	0

Figure 3.4: Solution using 3rd algorithm

X	0	10	2		X	6	6	6
3	X	1	12	+	6	X	6	6
4	1	X	12		6	6	X	6
11	1	7	X		6	6	6	X

Figure 3.5: The matrix after having 6 subtracted off

X	0	1	2		X	12	12	12
3	X	1	5	+	12	X	12	12
4	1	X	6		12	12	X	12
4	1	2	X		12	12	12	X

Figure 3.6: The matrix after the first multiplication

0	12	13	14		0	6	16	8
15	0	13	17	min	9	0	7	18
16	13	0	18		10	7	0	18
16	13	14	0		17	7	13	0

Figure 3.7: The matrix after performing addition

0	6	13	8
9	0	7	17
10	7	0	18
16	7	13	0

Figure 3.8: The final solution matrix

Chapter 4

Spectral Graph Theory

4.1 What Is Spectral Theory

Spectral graph theory essential is an expansion of the idea presented in section 3.3.1. The difference between the first idea is that under the idea of subtracting off the smallest number in the graph this graph has one range. Under spectral theory further investigation of the graph is required. So as to find more ranges rather than just the one which is used in the first.

Then the matrix is split into these sub-ranges and the smallest value in each range is subtracted off. This means that for the purposes of algorithm 3 from section 3.3 the largest number in the graph is the largest range length. These ideas can be displayed in the graph in Figure 4.1. This graph displays the frequencies of the edge costs contained in a graph. This sort of graph displays the ideal properties that a graph would want in order to take advantage of this new algorithm presented in this chapter.

4.2 New Algorithm - Spectral Analysis of Matrices

The outline for this new algorithm is as follows. The distance matrix is split into various sub matrices depending on which range the value falls into. Each value will fall into just one specific range as the example in Figure 4.2 shows, there are three ranges; range one is from 1 to 10; range two is from 45 to 55; range three is from 110 to 120. Figure 4.3 shows the distance matrix produced by retaining all the values for the first range and replacing all the values that fall into the other two ranges with infinity. The same thing has been done for the other two ranges and is shown in Figures 4.4 and 4.5.

Each range may then subtract the smallest value off so in the case of range one a 1 may be subtracted off, in the case of range two a 45 may be subtracted off and in the case of range three a 111 may be subtracted off. All of the smaller matrices now contain numbers that require less bits. Each of the graphs is multiplied with its self and with the other matrices. An offset matrix is constructed similar to the algorithm in the previous chapter. Each of the matrix multiplications is performed using algorithm 3 from section 3.3. The bound on

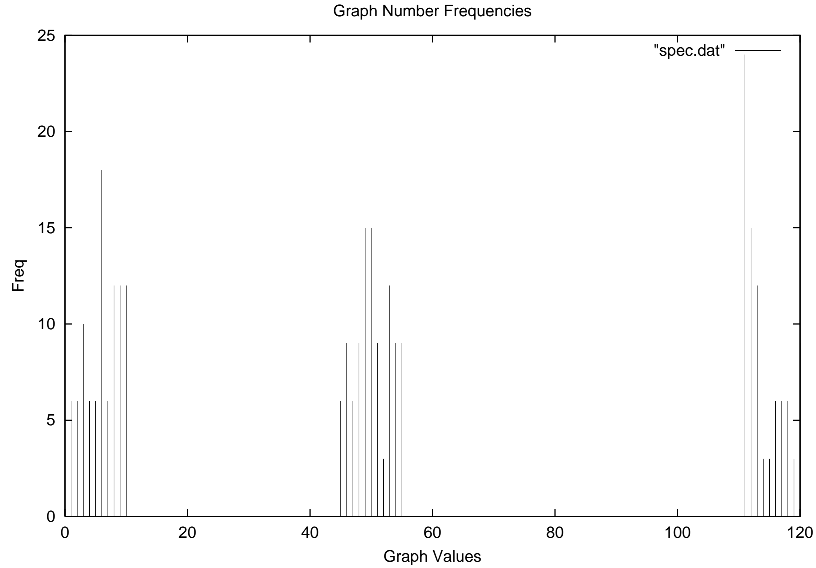

Figure 4.1: An example of a Good Spectral graph

1	47	4	6	112	111	46	49	51	111
46	119	54	10	8	6	116	3	52	112
54	118	50	45	46	9	115	8	46	113
4	5	116	111	49	111	1	48	49	6
111	48	8	3	113	55	10	113	50	117
50	53	9	49	50	53	118	5	54	8
51	7	51	53	49	111	3	10	111	9
113	2	3	6	8	2	112	55	46	120
9	111	45	117	48	47	120	112	53	50
6	10	120	112	6	118	7	3	114	55

Figure 4.2: An example of a distance matrix

the maximum edge cost for the algorithm from section 3.3 was $0(n^{0.624})$, this bound is now no longer the size of the graph but now is the maximum range length.

The repeated squaring of the matrix, which has 2 ranges, looks like this $(A + D_1 \ min \ B + D_2)(A + D_1 \ min \ B + D_2)$. This is expanded to $(A^2 + 2D_1)min(AB + D_1 + D_2)min(BA + D1 + D2)min(B^2 + 2D_2)$.

4.3 A Problem With This New Algorithm

A problem with using this algorithm to solve the all pairs shortest path is that there each band can double in size after the multiplication. When using repeated squaring of the matrix and doing the distance matrix multiplication by using this method, then each time the matrix is multiplied the size of each band could

1	-	4	6	-	-	-	-	-	-
-	-	-	10	8	6	-	3	-	-
-	-	-	-	-	9	-	8	-	-
4	5	-	-	-	-	1	-	-	6
-	-	8	3	-	-	10	-	-	-
-	-	9	-	-	-	-	5	-	8
-	7	-	-	-	-	3	10	-	9
-	2	3	6	8	2	-	-	-	-
9	-	-	-	-	-	-	-	-	-
6	10	-	-	6	-	7	3	-	-

Figure 4.3: Sub-Range one

-	47	-	-	-	-	46	49	51	-
46	-	54	-	-	-	-	-	52	-
54	-	50	45	46	-	-	-	46	-
-	-	-	-	49	-	-	48	49	-
-	48	-	-	-	55	-	-	50	-
50	53	-	49	50	53	-	-	54	-
51	-	51	53	49	-	-	-	-	-
-	-	-	-	-	-	-	55	46	-
-	-	45	-	48	47	-	-	53	50
-	-	-	-	-	-	-	-	-	55

Figure 4.4: Sub-Range two

-	-	-	-	112	111	-	-	-	111
-	119	-	-	-	-	116	-	-	112
-	118	-	-	-	-	115	-	-	113
-	-	116	111	-	111	-	-	-	-
111	-	-	-	113	-	-	113	-	117
-	-	-	-	-	-	118	-	-	-
-	-	-	-	-	111	-	-	111	-
113	-	-	-	-	-	112	-	-	120
-	111	-	117	-	-	120	112	-	-
-	-	120	112	-	118	-	-	114	-

Figure 4.5: Sub-Range three

have doubled. So the bands could very quickly merge together as one and then you would no longer be able to use this algorithm. This can be visualised by firstly looking at the graph in Figure 4.1 which shows 3 ranges of numbers, the next graph, see Figure 4.6, shows what could have happened to that same graph after one matrix multiplication. Each of the ranges has spread out and with further multiplications the ranges would become more and more bunched together.

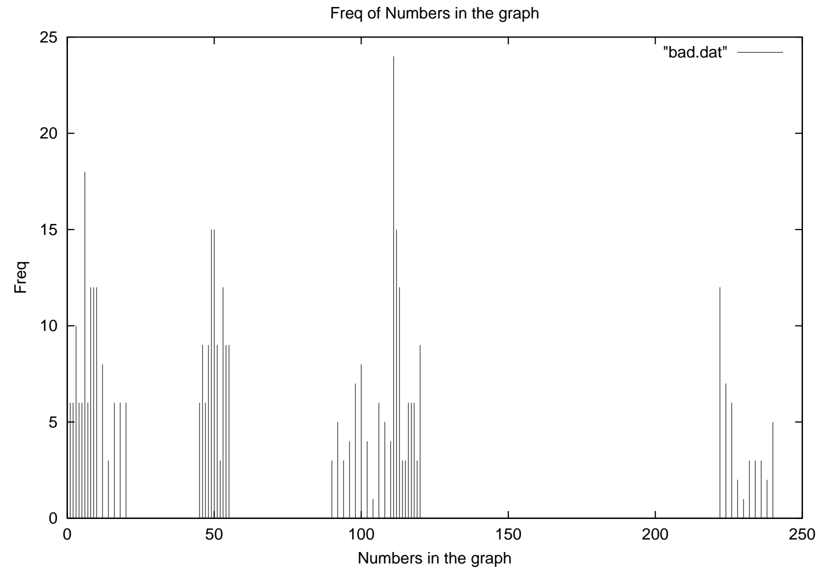

Figure 4.6: An example of What may happen to the graph

Chapter 5

Two-band Algorithm

Expanding on these graph concepts presented in the previous chapters is the 2-band modified Floyd algorithm. This algorithm takes a two band approach, where as the algorithm from the previous chapter took a n band approach. Under this approach the graph consists of small sub-graphs that are then connected to each other to make up the entire graph. This can be demonstrated in Figure 5.2. If the reader looks at this figure they would be able to see that the graph consists of four sub-graphs, that are interconnected with each other. A requirement for each sub-graph is discussed in section 5.1.

Each of these individual subgraphs can then be calculated using Floyd's algorithm. then the overall graph can be calculated using a modified Floyd's algorithm. This algorithm is presented in the next section. Floyd's algorithm was presented in the second chapter of this report.

5.1 Requirement For the Sub-graph

Each node must have a path which has a specific length between each other node. This means that after each single node has been calculated the solution matrix for this sub-graph must have all values falling within a specific range.

5.2 The Two-band Algorithm

For a graph that consists of k sub-graphs each of the k sub-graphs is calculated using Floyd's algorithm presented in the first chapter. This is displayed below in the next piece of code. The algorithm is just Floyd's algorithm except for the first line which represents the algorithm being run over the k sub-graphs. This part of the algorithm has a time complexity of $O(\frac{n}{k}k^3)$, this is reduced to $O(nk^2)$.

```
1 for p = 1 to n/k
2   for i = 1 to k
3     for j = 1 to k
4       for l = 1 to k
5         M[i][j] = M[i][j] min
            (M[i][l] + M[l][j]);
```

This next piece of code shows the modified Floyd algorithm. This algorithm is basically the same as the original Floyd's algorithm, but in line 4 a matrix multiplication is performed instead of an addition. The time complexity of this algorithm is $O(\frac{n^3}{k^3})$ for the first 3 lines and line 4 requires time complexity of $O(k^3)$ for the matrix multiplication. When these two are combined and reduced an overall time complexity of $O(n^3)$ is found. This second part of the algorithm is more dominant than the first part of the algorithm and so the first part of the algorithm can be absorbed into the second.

```
1 for i = 1 to n/k do
2   for j = 1 to n/k do
3     for l = 1 to n/k do
4       Matrix[i][j] = Matrix[i][j] min
         Matrix[i][l] * Matrix[l][j];
```

This leaves the algorithm having the same time complexity as that of Floyd's algorithm. A way of decreasing the time complexity of this algorithm would be to subtract the minimum value off of the sub-graphs and use algorithm 3 from section 3.3. If this algorithm is used for the multiplication then a decrease in the time complexity could be achieved due to the fact that the numbers in the sub-graph would be using less bits.

Algorithm 3 had a relationship based on the size of the largest edge cost called m. line 4 can be performed in $O(f(m,k)$. This would then lead to an overall time complexity of $O(f(m,k)(\frac{n}{k})^3)$. As long as the function f is kept below k^3 then a decrease in time can be achieved by this algorithm.

5.3 Deciding How To Split The Subgraphs

Splitting these bands into the various sub-graphs is not an easy task. Mathematically it must be such that each sub-graph must only go from one point in the sub-graph to every other point in the sub-graph without using an edge that exists outside of the sub-graph. If M is the length of the longest path in this sub-graph then all edges leaving or entering this sub-graph must have a value greater than $M \div 2$. There is no algorithm for finding these sub-graphs without using to much time up so the sub-graphs that the author was working with had sub-graph longest path of the $\sqrt{n}$. The length of the shortest path leaving this sub-graph was n. These points raised in this section can be demonstrated in figure 5.1.

5.4 Predictable Gain

Comparing this algorithm's time complexity with that of say Floyd's for graphs where the sub-graph has largest edge cost of $\sqrt{n}$ has a time complexity the same as that of Floyd's, Floyd's has time complexity of $O(n^3)$ and this algorithm has time complexity of $O(n^3)$ as discussed in section 5.2.

If an approach is taken to reduce the costs of the individual matrices and then use algorithm 3 again it is not possible to show time complexity results due to the fact that an increase in speed only becomes apparent when dealing with numbers consisting of 1000 bits.

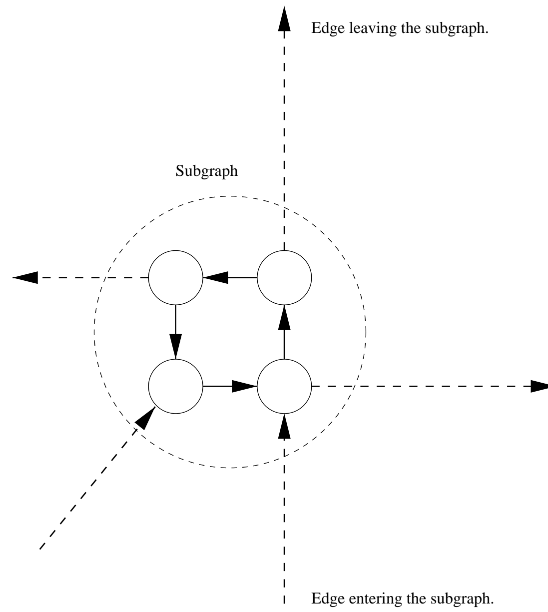

Figure 5.1: Demonstrating how the subgraphs are split

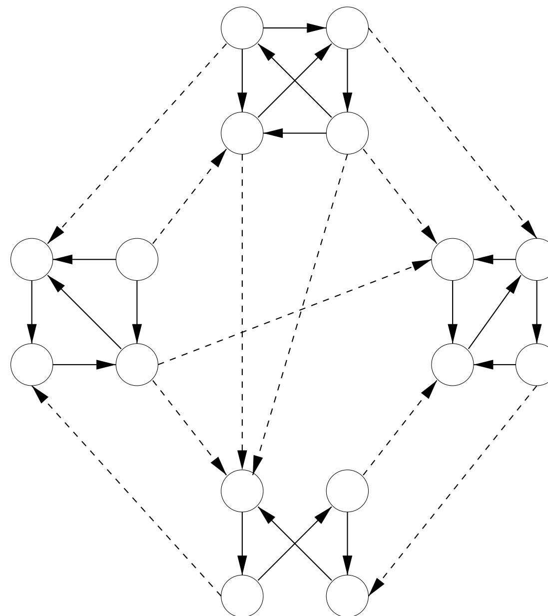

Figure 5.2: An example graph structure

Chapter 6

Maximum Sub-Array problem - An application of The All Pairs Shortest Path

As an aside to the work from the previous chapters. The maximum sub array problem, which involves maximising the negative and positive numbers in a matrix, so as to choose that part of the matrix which has the largest sum in it. This is trivial in a matrix that contains only positive numbers as naturally it will be the sum of the whole matrix. But when that matrix contains negative and positive numbers then another kind of algorithm will have to be used. The brute force algorithm for solving this has a time complexity of $O(n^4)$. Recently this problem was shown to be equivalent to the all pairs shortest path problem. This algorithm is therefore a perfect example of another good use for the all pairs shortest path algorithms. The maximum sub-array problem can be used as an efficient algorithm for data mining of information and for identifying the brightest portion of an image.

6.1 Maximum Sub Array Algorithm

This algorithm is a divide and conquer type algorithm and only the outer frame is presented, for an in depth look at the algorithm see the paper by T. Takaoka [11] or the paper by H. Tamaki and T. Tokuyama[12].

The Algorithm selects the overall solution from the maximum of the six solutions given below. The solutions for the NW,SW,NE,SE parts of the matrix are given by recursively calling the algorithm on these parts. The row centred problem and the column centred problem is when the maximum sub-array crosses either the vertical or horizontal dividing lines. The algorithms for solving both these problems is given in the paper by T. Takaoka[11].

- A_{nw} Solution for the NW part of the matrix.
- A_{sw} Solution for the SW part of the matrix.
- A_{ne} Solution for the NE part of the matrix.

- A_{se} Solution for the SE part of the matrix.
- A_{row} Solution for the row-centred problem.
- A_{column} Solution for the column-centred problem.

This is illustrated in the diagram that shows the matrix split into its various parts. This diagram is found in figure 6.1.

The relationship between this problem and the all pairs shortest path problem is in the solution to the column centre and row centre. Both these can be reduced to the all pairs shortest path problem.

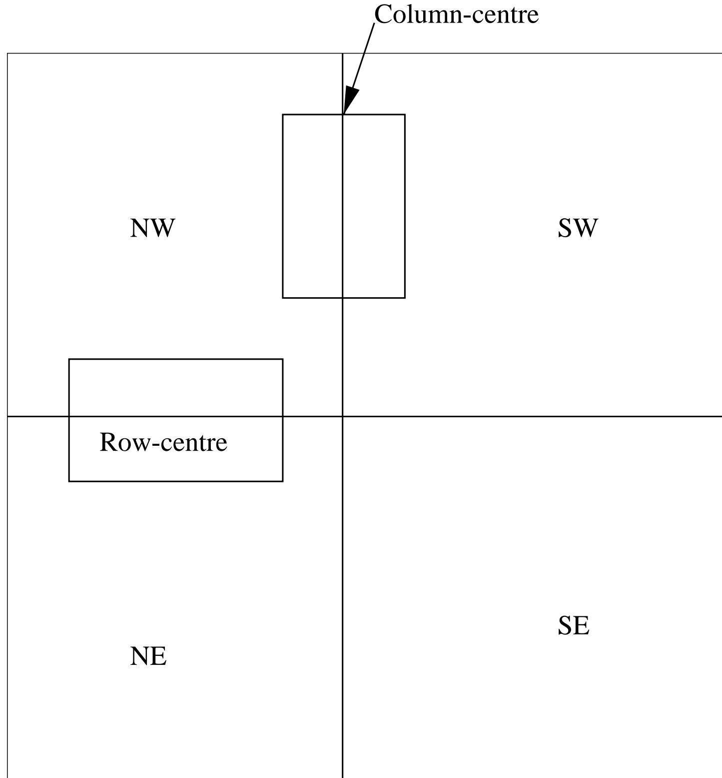

Figure 6.1: Diagram illustrating the splitting of the matrix

6.2 Results

Takaoka made an improvement to this initial algorithm. The results from Takaoka's algorithm and the original brute force algorithm can be seen on the graph in Figure 6.2. The two algorithms were timed on graphs containing 16,32,64,128 and 256 nodes. The time taken to calculate the maximum subarray was then recorded for each of the two algorithms and this was plotted on the graph. Takaoka's algorithm was using Floyd's algorithm but if others with smaller time complexity were used then better results could be achieved.

Their is further work required in this field to try different All pairs shortest path algorithms for this maximum sub array problem and test to see which are more effective.

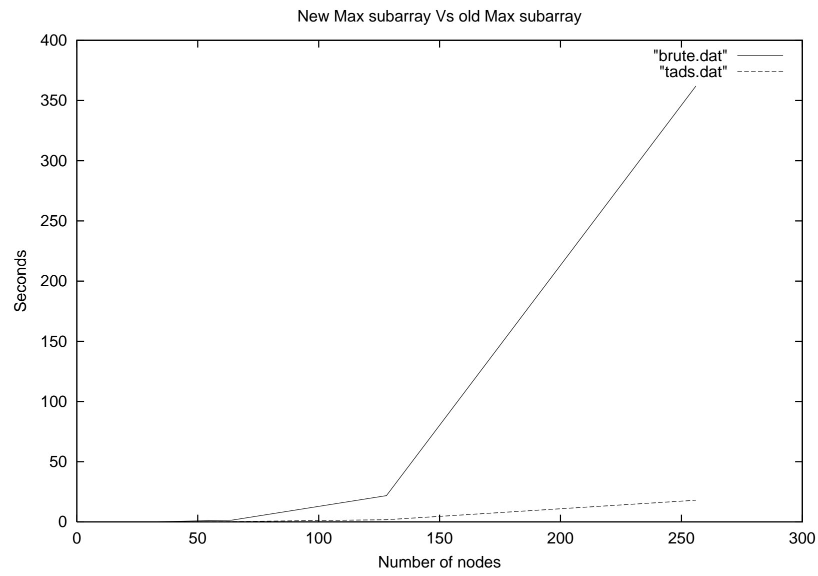

Figure 6.2: Results for the maximum sub array problem

Chapter 7

A Graphical Front End For Displaying Graph Algorithms

A Graphical user interface for displaying these graph algorithms at work is under development. This would allow the user the ability to display a graph algorithm running and to show the path calculated by the all pairs shortest path algorithm that user has selected to display. Currently this is written in the TCL/TK language with an adaption to the TCL interpretor so as to allow it to handle matrices. The Figure 7.1 displays a screen-shot of the graphical front end. In this you can see a graph entered into the canvas widget. There are large buttons to select either nodes or lines, which the user could then draw onto the canvas widget. A user can select a line and then type in the entry box a value for that edge. When the multiply button is hit the program goes off and calculates the all pairs shortest path problem. This program uses a different algorithm depending on what the user has selected from the *Algorithm* menu.

7.1 Future Development

This project started out as a small aside from the main project and as such has not been fully completed. There is a lot more work that could be put into this project in the future. Developing this in the future would require changing this to a web based approach and displaying the results of the algorithm running at each step and then the final overall results. This program would also have many more graph algorithms added to it, such as single source algorithms etc.

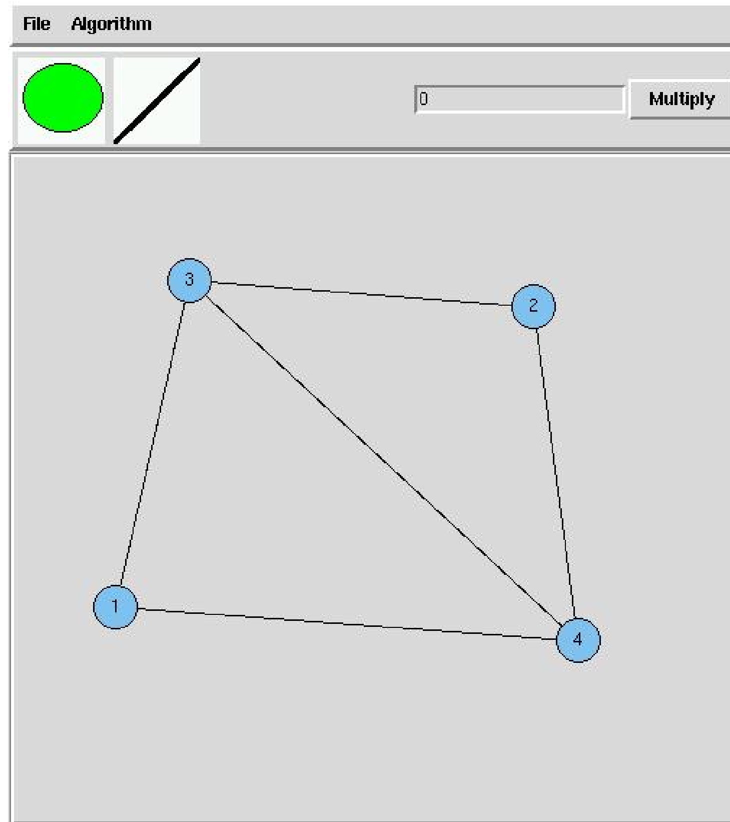

Figure 7.1: Screen-shot from the graphical front end

Chapter 8

Conclusion

In this report the author has presented the user with a number of new approaches to deal with the all pairs shortest path algorithm. These new approaches are to do with looking at the structure of the graph and then using this information so as to gain further decreases in time complexity.

The approach taken in this project was to take the original algorithm, 3rd algorithm presented in Takaoka's paper [6]. The first approach found that you could subtract off the minimum value from the matrix and therefore benefit from a very small decrease in the time complexity.

Then we went on to look at spectral graph theory, which involves splitting the graph into various bands or ranges of numbers, separating the graph into many separate graphs consisting of these bands, using a modified algorithm of algorithm 3 and finally putting the whole thing back together as the solution matrix. In this approach you gain efficiency from the subtracting off of the lower bound for each range. This algorithm however had a draw back as discussed in section 4.3.

Finally the Algorithm from chapter 5 was discovered which is again further refinement of the work done in the previous chapter. This algorithm was able to take graphs with a specific structure and use this structure so as to gain a decrease in the time complexity of the algorithm by making use of the algorithm 3 discussed in chapter 3.

So in concluding many authors looking into the problem of the all pairs shortest path have come up with a number of algorithms that deal with the structure of the graph, these algorithms were however dealing with things such as sparse graphs and acyclic graphs. Not much work has been done actually looking at the overall edge costs and how these edge costs interact. There is certainly a lot more work in this field of spectral analysis of graphs for another researcher to look into.

8.1 Further Work

Additional work that could be done is this area could include, looking further into the splitting of the graph into its various sub-graphs this concept comes from chapter 5. Further work needs to be performed in order to find an algorithm for finding these various sub-graphs. As discussed at the end of chapter 7 the

graphical front end has a lot of further work that could be performed on it.

Appendix A

Code listings

```
#include "strassen.h"
#include "tad.h"
#include <time.h>
#define MAX 1000

int **A,**B;
int **sol1,**sol2;

int makematrix(int n,double p) {
    int i,j;                                                    10
    double temp;

    for(i=0;i < n;i++) {
        for(j=0;j < n;j++) {
            temp = (double) ((rand() % MAX) + 1);
            temp = temp / (double) MAX;
            if(temp >= p) {
                A[i][j] = 2;
            } else {
                A[i][j] = INF;                                  20
            }
        }
        A[i][i] = 1;
        if(i != (n-1)) {
            A[i][i+1] = 2;
        } else {
            A[i][0] = 2;
        }
    }
    return 0;                                                   30
}

int printmatrix(int **A,int n) {
    int i,j;
    printf("\n");
    for(i=0;i<n;i++) {
        for(j=0;j<n;j++) {
            printf("%4i",A[i][j]);
        }
        printf("\n");                                           40
    }
    printf("\n");
    return 0;
}

int main(int argc,char *argv[]) {
    time_t t;
    clock_t c;
    int n,i,j;
    double p;                                                   50

    if(argc != 3) {
        fprintf(stderr,"%s int double\n",argv[0]);
        exit(1);
    }
    if ((n = atoi(argv[1])) == 0) {
        fprintf(stderr,"The first parameter must be an integer\n");
        exit(1);
    }
    if ((p = atof(argv[2])) == 0) {                             60
        fprintf(stderr,"The second parameter must be a double\n");
        exit(1);
    }
    sol1 = M_Matrix(n,INF,INF);
    sol2 = M_Matrix(n,INF,INF);
```

```
#include "strassen.h"

int ** M_Matrix(int n,int one, int two) {
    int i,j;
    int **temp;

    temp = malloc(sizeof(int*) * n);

    if(temp == NULL) {
        perror("Malloc Failed\n");                                   10
        exit(1);
    }

    for(i=0;i<n;i++) {
        temp[i] = calloc(n,sizeof(int));
        if(temp[i] == NULL) {
            perror("Malloc Failed\n");
            exit(1);
        }
        if(two != -1) {                                              20
            for(j=0;j<n;j++) {
                temp[i][j] = two;
            }
        }
        if(one != -1) {
            temp[i][i] = one;
        }
    }

    return temp;                                                     30
}

void free_m(int **t,int n) {
    int i;

    for(i=0;i<n;i++) {
        free(t[i]);
    }
    free(t);
}                                                                    40


int strassen (int **solution,int **A, int **B, int n) {
    int halfn = n/2;
    int **M1,**M2,**M3,**M4,**M5;
    int **M6,**M7,**Anew,**Bnew;
    int i,j;

    M1 = M_Matrix(halfn,-1,-1);
    M2 = M_Matrix(halfn,-1,-1);                                      50
    M3 = M_Matrix(halfn,-1,-1);
    M4 = M_Matrix(halfn,-1,-1);
    M5 = M_Matrix(halfn,-1,-1);
    M6 = M_Matrix(halfn,-1,-1);
    M7 = M_Matrix(halfn,-1,-1);
    Anew = M_Matrix(halfn,-1,-1);
    Bnew = M_Matrix(halfn,-1,-1);

    if(n == 2) {
        M1[0][0] = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]);        60
        M2[0][0] = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);
        M3[0][0] = (A[0][0] - A[1][0]) * (B[0][0] + B[0][1]);
        M4[0][0] = (A[0][0] + A[0][1]) * B[1][1];
        M5[0][0] = A[0][0] * (B[0][1] - B[1][1]);
        M6[0][0] = A[1][1] * (B[1][0] - B[0][0]);
```

```
#include "tad.h"
#include "strassen.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

void test() {
    return;
}

double logb(double x,double b) {
    return log(x) / log(b);
}

int pmatrix(int **A,int n) {
    int i,j;
    printf("\n");
    for(i=0;i<n;i++) {
        for(j=0;j<n;j++) {
            printf("%4i",A[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    return 0;
}

int round(double x) {
    int i;

    i = (int) x;
    x = x − (double) i;
    if(x >= 0.5) {
        i = i  + 1;
    }
    return i;
}

int A_G_M(int **D,int **A, int n,float r) {
    int l,i,j;
    int **B,**C;
    int flag;

    flag = 0;
    B = M_Matrix(n,1,−1);
    C = M_Matrix(n,−1,−1);

    for(l=2; l <= (int) r;l++) {
        if(flag == 0) {
            strassen(C,A,B,n);
            for(i=0; i < n; i++) {
                for(j=0;j<n;j++) {
                    if(C[i][j] > 1) {
                        C[i][j] = 1;
                    }
                }
            }
        } else {
            strassen(B,A,C,n);
            for(i=0; i < n; i++) {
                for(j=0;j<n;j++) {
                    if(B[i][j] > 1) {
                        B[i][j] = 1;
                    }
                }
```

```
#define MAX 512
#define INF 999

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

typedef int Matrix_t[MAX][MAX];

Matrix_t a,b,u,v,w,s,t,sum,column,temp;

void subarray(int m1,int m2,int n1,int n2,int *value);
void mult(int a,int b,int c,int d,int *value);
int max(int x1,int x2,int x3,int x4,int x5,int x6);


void mult(int a,int b,int c,int d,int *value) {
  int i,j,k,l,m,n;
  Matrix_t sum1;

  m = b-a+1;
  n = d-c+1;
  for(i=1;i<=m;i++) {
    for(j=1;j<=n;j++) {
      sum1[i][j] = temp[a+i-1][c+j-1];
    }
  }
  for(i=1;i<=n;i++) {
    for(j=1;j<=n;j++) {
      s[i][0] = 0;s[0][j] = 0;
      t[i][0] = 0;t[0][j] = 0;
      s[i][j] = sum1[i][j];
      t[i][j] = -sum1[j][i];
    }
  }
  for(i=1;i<=m;i++) {
    for(k=0;k<=m;k++) {
      w[i][k] = INF;
      for(l=0;l<=(n / 2 - 1);l++) {
        if((s[i][l] + t[l][k]) < (w[i][k])) {
          w[i][k] = s[i][l] + t[l][k];
        }
      }
    }
  }
  for(i=1;i<=m;i++) {
    for(j=1;j<=m;j++) {
      if(i<=j) {
        w[i][j] = INF;
      }
    }
  }

  for(i=1;i<=m;i++) {
    for(j=(n/2+1);j<=n;j++) {
      v[i][j] = INF;
      for(k=0;k<=m;k++) {
        if((w[i][k] + s[k][j]) < (v[i][j])) {
          v[i][j] = w[i][k] + s[k][j];
        }
      }
    }
  }
  for(i=1;i<=m;i++) {
```

```
#!/users/cosc/honours/djc110/Cosc460/tixScript/mytcl -f
set menuframe .menufr
set filebut $menuframe.file
set algobut $menuframe.algo
set fileMenu $filebut.f
set algoMenu $algobut.f
set mainframe .mainfr
set iconframe .iconfr
set myc $mainframe.mycanvas
set butCircle $iconframe.cir
set butMul $iconframe.mul
set butLine $iconframe.line
set textbx $iconframe.txt
set Matrix [NewMatrix]
set node 0
set cir 1
set line 2
set currline(val) 0
set currline(xval) 0
set currline(yval) 0
set currline(xlast) 0
set currline(ylast) 0
set currnode 0
set OPT(width) 500
set OPT(height) 460
set OPT(mnhgt) 40
set OPT(cirfile) "~djc110/Cosc460/tixScript/cir.gif"
set OPT(linefile) "~djc110/Cosc460/tixScript/line.gif"
set OPT(SaveFile) 0
set OPT(LoadFile) 0
set number 0
set NODES 0
set Row 0
set Col 0

frame $menuframe -width $OPT(width) -height $OPT(mnhgt) -relief raised -bd 3
frame $mainframe -width $OPT(width) -height $OPT(height)
frame $iconframe -width $OPT(width) -height $OPT(mnhgt) -relief raised -bd 3
menubutton $filebut -text "File" -menu $fileMenu
menubutton $algobut -text "Algorithm" -menu $algoMenu
menu $fileMenu
menu $algoMenu
image create photo $cir -file $OPT(cirfile)
image create photo $line -file $OPT(linefile)
button $butCircle -image $cir -command "Nodes $myc" -bd 1 -relief flat
button $butLine -image $line -command "Lines $myc" -bd 1 -relief flat
button $butMul -command "runMul" -text "Multiply"
canvas $myc -relief groove -bd 3 -width $OPT(width) -height $OPT(height)
entry $textbx -textvariable number -state disabled

bind $textbx <KeyPress-Return> "TextEnter"

$fileMenu add command -label "New" -command {New}
$fileMenu add command -label "Save File ..." -command {Save}
$fileMenu add command -label "Load File ..." -command {Load}
$fileMenu add command -label "Quit" -command {exit}
```



```
proc New {} {
    global Matrix node
    set i 1
    while {$i <= [SizeMatrix $Matrix]} {
        $myc delete [getNode $Matrix $i]
        incr i
```

```
#include <tk.h>
#ifndef MATRIX_H
#define MATRIX_H

void init_Matrix(int n);

int NewMatrix(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]);
                                                    10
int delMatrix(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]);

int setMatrix(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]);
int getMatrix(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]);              20

int SizeMatrix(ClientData clientData,
             Tcl_Interp *interp,
             int argc,char *argv[]);

int setNodePos(ClientData clientData,
             Tcl_Interp *interp,
             int argc,char *argv[]);

int getNodePos(ClientData clientData,               30
             Tcl_Interp *interp,
            int argc,char *argv[]);

int getNodeVal(ClientData clientData,
             Tcl_Interp *interp,
            int argc,char *argv[]);

int setNodeVal(ClientData clientData,
             Tcl_Interp *interp,
            int argc,char *argv[]);                 40

int getLines(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]);

int getNode(ClientData clientData,
            Tcl_Interp *interp,
            int argc,char *argv[]);

int getLine(ClientData clientData,Tcl_Interp *interp,   50
            int argc, char *argv[]);

int setLine(ClientData clientData,Tcl_Interp *interp,
            int argc, char *argv[]);

int MatrixMul(ClientData clientData,Tcl_Interp *interp,
            int argc, char *argv[]);

#endif
```


Figure A.6: Source Code for matrix.h

```
int writeout(int n,int r,int *a,int **m,char *f);
int inrange(int n);
int center(int c,int n);
int ** makematrix(int n,int r,int *ranges,float *weights);
int printmatrix(int **matrix,int n);
```

Figure A.7: Source Code for matrix.h

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include "matrix.h"
#include "create.h"

int writeout(int n,int r,int *a,int **m,char *f) {
  FILE *ptr;
  int i;

  if(strcmp(f,"stdout") == 0) {
    ptr = stdout;
  } else {
    ptr = fopen(f,"w");
    if(ptr == NULL) {
      return -1;
    }
  }

  fwrite(&n,sizeof(int),1,ptr);
  fwrite(&r,sizeof(int),1,ptr);
  fwrite(a,sizeof(int),r,ptr);
  for(i=0;i < n;i++) {
    fwrite(m[i],sizeof(int),n,ptr);
  }

  return 0;

}

int inrange(int n) {
  float c,r;
  c = 1 / (float) n;
  r = (float) (random() % 100000) / 100000.0;
  return (int) (r / c);
}

int center(int c,int n) {
  int number;

  number = inrange(n);
  if(inrange(2)) {
    number = 0 - number;
  }
  return c + number;
}

int ** makematrix(int n,int r,int *ranges,float *weights) {
  int **matrix;
  int i,j;

  matrix = newMatrix(n);

  for(i=0;i<n;i++) {
    for(j=0;j<n;j++) {
      matrix[i][j] = center(ranges[inrange(r)],10);
    }
  }
  return matrix;
}

int printmatrix(int **matrix,int n) {
  int i,j;
```

```
#ifndef CREATE_H
#define CREATE_H
int ** newMatrix(int n);
#endif
```

Figure A.9: Source Code for create.h

```
#ifndef SPEC_H
#define SPEC_H

int setup(int n,int r,int **M);

int ** spectral(int n,int r,int *rang,int **M);

int ** normal(int n,int **M);
                                                                        10
int isrange(int n,int a,int l);

int divide(int n,int r,int *rang,int **M);

int multiplySpec(int n,int i,int j,int r,int *rang,int **M,int l);

int check(int n,int **spec,int **nor);
#endif
```

Figure A.10: Source Code for spec.h

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  FILE *ptr;
  int n,r,*range;
  int **M,i,j;


  ptr = fopen(argv[1],"r");

  fread(&n,sizeof(int),1,ptr);
  fread(&r,sizeof(int),1,ptr);
  range = (int *) malloc(sizeof(int) * r);
  fread(range,sizeof(int),r,ptr);
  M = (int **) malloc(sizeof(int*) * n);
  for(i=0;i<n;i++) {
    M[i] = (int*) calloc(n,sizeof(int));
    fread(M[i],sizeof(int),n,ptr);
  }

  printf("\nNodes: %i ",n);
  printf("Ranges %i\n",r);
  for(i=0;i<r;i++) {
    printf("%4i ",range[i]);
  }
  printf("\n");
  for(i=0;i<n;i++) {
    for(j=0;j<n;j++) {
      printf("%4i ",M[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Figure A.11: Source Code for testm.c

Bibliography

[1] F. Romani. Shortest-path problem is not harder that matrix multiplications. *Inform. Process. Lett.*, 11:134–136, 1980.

[2] Tadao Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing letters*, 43(6):195–199, September 1992.

[3] T. Takaoka. Shortest path algorithms for nearly acyclic directed graphs. *Theoretical Computer Science*, 203:143–150, 1998.

[4] A. Moffat and T. Takaoka. An all pairs shortest path algorithm with expected time $o(n^2 log\ n)$. *SIAM J. Comput.*, 16:1023–1031, 1987.

[5] R.W. Floyd. Algorithm 97: Shortest path. *Comm. ACM*, 5:345, 1962.

[6] Tadao Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 20:309–318, 1998.

[7] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *Proc. 32nd IEE FOCS*, pages 569–575, 1991.

[8] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9:251–280, 1990.

[9] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:334–6, 1969.

[10] A. Schonhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing.*, 7:281–292, 1971.

[11] T. Takaoka. An efficient algorithm for data mining based on the maximum subarray problem. 1999.

[12] H. Tamaki and T. Tokuyama. Algorithms for the maximum subarray problem based on matrix multiplication. *Proceedings of the 9th SODA(Symposium on Discrete Algorithms).*, pages 446–452, 1998.

[13] M.L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5:83–89, 1976.