INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Pagenumber Problem

A thesis
presented to the University of Ottawa
in partial fulfillment of the
requirements towards the award of the degree in
Master of Computer Science*

by

(C) Nidhi Kapoor

School of Information Technology and Engineering,
University of Ottawa,
Ottawa,
Ontario, Canada

* The Masters of Computer Science program is offered through the School of Computer Science at Carleton University and the School of Information Technology and Engineering at the University of Ottawa

Acknowledgements

I am greatly obliged to my supervisor, Dr. Ivan Stojmenovic, who proposed the problem and gave valuable suggestions that have improved this thesis and its presentation. I would also like to thank him for his patience and support throughout.

I am thankful to Dr. Albert Zomaya (University of Western Australia), for research related to the problem.

I am very grateful to Alioune for all his guidance, advice, and helpful discussions.

I would also like to thank Mark Russell for his contribution to the results presented in chapter 6.

Abstract

A " book-embedding" of a graph G comprises of embedding the graph's nodes along the spine of a book, and embedding the edges on the pages so that the edges embedded on the same page do not intersect. This is also referred to as the page model. The "pagenumber" of a graph is the thickness of the smallest (in number of pages) book into which G can be embedded.

A literature review of the one dimensional pagenumber problem is presented, and several two dimensional pagenumber models are proposed. Evolutionary computing methods on problems whose solution space comprises of permutations are reviewed. Since the pagenumber problem is known to be NP-complete, we describe two solutions using Hill Climbing methods and one solution using Genetic Algorithms for one and two-dimensional models. Two two- dimensional models are considered namely the square and rook models.

We have given a unified framework for all three pagenumber models, in which a solution is a pair of two permutations (of nodes and edges), and which differ only by criteria for edge intersections. Experimental results on several kinds of graphs are then given.

Contents

1. INTRODUCTION

1.1 The pagenumber problem

We commence with some important definitions: a *graph G (V, E)* comprises of a set of vertices *V*, and a set of undirected edges *E* connecting the vertices. A *book* consists of a spine and some number of *pages* (each a half-plane with the spine as boundary). A *book embedding* of a graph is a linear ordering of the vertices along the spine of a book and an assignment of edges to pages so that edges assigned to the same page can be drawn on that page without crossing. The minimum number of pages in which a graph can be embedded is its *pagenumber*.

The pagenumber problem is hard; that is, for a given linearization of the nodes of a graph *G*, and a given integer *k*, the problem of deciding if the linearization admits a *k*-page book embedding of *G* is NP-complete [GJMP]. The problem is relevant in several realms, some of which are described in section 1.2.

Consider a 2-D model for embedding the edges of a graph. Here, *n* nodes are placed on a plane *m*, and each page corresponds to a planar graph. The problem is to minimize the number of pages. This model has not been studied before.

1.2 Applications of pagenumber problem

Direct Interconnection Networks. The pagenumber problem is relevant in the VLSI implementation of interconnection networks, in obtaining the optimal layout for processors and wires to be interconnected. With respect to interconnection networks, the vertices of a graph correspond to processors, while the edges between vertices correspond to links between processors. The figure below shows a four-vertex graph and its corresponding two-page embedding.

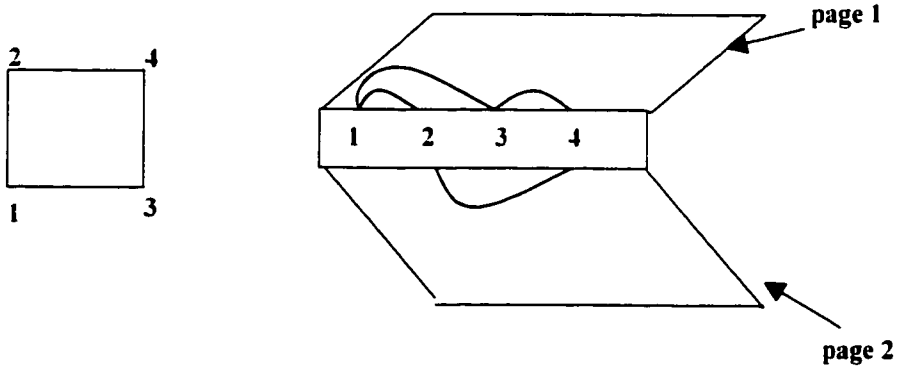


Fig. 1.1 Two page embedding of a square

The next figure shows a one-page embedding of the same graph by embedding a different permutation of the vertices on the spine of the book:

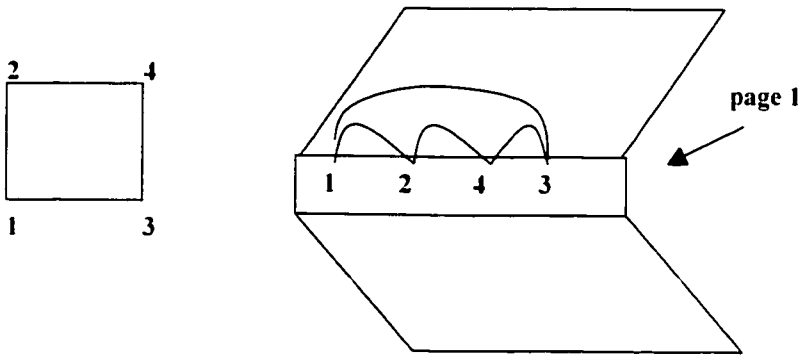


Fig. 1.2 One page embedding of a square

Since the permutation of vertices (1243) yields a better (smaller) pagenumber as compared to the permutation of vertices (1234), we can state that the permutation (1243) is an optimal layout for the graph in figure 1.

Fault-tolerant processor arrays. Another application is in fault tolerant VLSI design [CLR1], [CLR2]. This application is referred to as DIOGENES. Processors are arranged (physically or logically) on a line and bundles of wires with embedded switches run parallel to the line of processors. Each bundle is capable of implementing a hardware stack of connections among processors. Each connection occurs on exactly one hardware stack (bundle). The line of processors is scanned from left to right and suppose that a good processor u wants to communicate with another processor v to its right. Then, at u, the connection (u, v) is pushed into one of the stack bundles, that is, (u, v) occupies the bottom of the stack bundle, while the other connections that are currently in this bundle are shifted up one place. At v, the connection (u, v) is popped from the stack. The problem is to realize the desired interconnection graph using the minimum number of stack bundles. This is equivalent to the problem of embedding the graph in a book with the minimum number of pages. The pages correspond to bundles. The constraint that edges on the same page do not intersect corresponds to the LIFO property of stacks.

Sorting with parallel stacks. Even and Itai [EI] studied the problem of how to realize fixed permutations of $\{1,\ldots,n\}$ with non-communicating stacks. Initially, each number is pushed in the order 1 to n, onto any one of the stacks. After all the numbers are on stacks, the stacks are popped to form the permutation. This problem can be formulated as follows. Consider the bipartite graph G_n with vertices $\{a_1,\ldots a_n, b_1,\ldots,b_n\}$ and edges connecting each a_i to b_i. The problem of realizing the permutation π on $\{1,\ldots,n\}$ with k parallel stacks is equivalent to embedding G_n in a k-page book, with its vertices embedded in the order $a_1,\ldots a_n, b_{\pi(1)},\ldots,b_{\pi(n)}$.

Single-row routing. In an attempt to simplify the problem of routing multilayer printed circuit boards (PCBs), H.C. So [S] decomposed the problem in the following way. In his variant method, the circuit elements are arranged in a regular grid, with wiring channels separating rows and columns of elements. The circuit's net lists are then decomposed (possibly by adding new dummy elements) so that every net connects elements in a single row or in a single column. The PCB can now be routed by routing each of its rows and each of its columns independently. The variant of this scenario that does not allow a net to run from the top of a row around to its bottom nor to change layers en route [RS] corresponds directly to the embedding problem applied to small-valence graphs.

Ordered sets. In some instances, the underlying structure may be an ordered set [NP]. For example, the vertices of the graph may represent steps in a calculation and the edges may represent necessary inputs for that calculation to be executed. If the calculation is done on a machine with a single processor, then each page represents a stack, and the edges indicate the order in which the partial results are entered and removed from the stack. The pagenumber is the smallest number of stacks required to store data in order for the calculation to be carried out. As an example, consider the ordered set represented by the graph in figure 1.3.

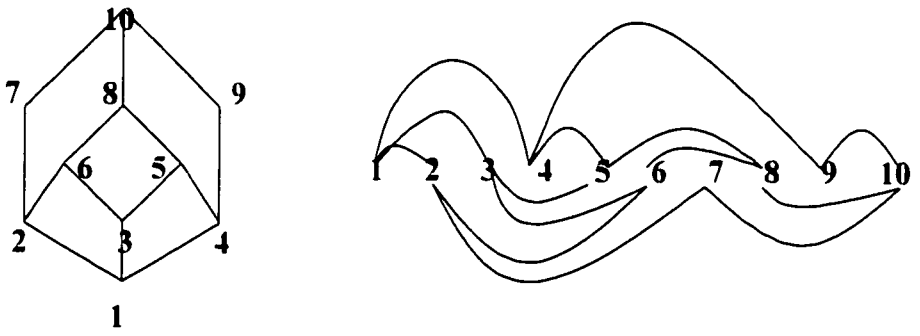


Fig. 1.3 Graph representing ordered set with its two-page embedding

The linear extension (1 2 3 4 5 6 7 8 9 10) yields a pagenumber of 2, while the linear extension (1 4 2 3 5 6 8 9 7 10) yields a pagenumber of 3, as shown in figures 1.3 and 1.4, respectively.

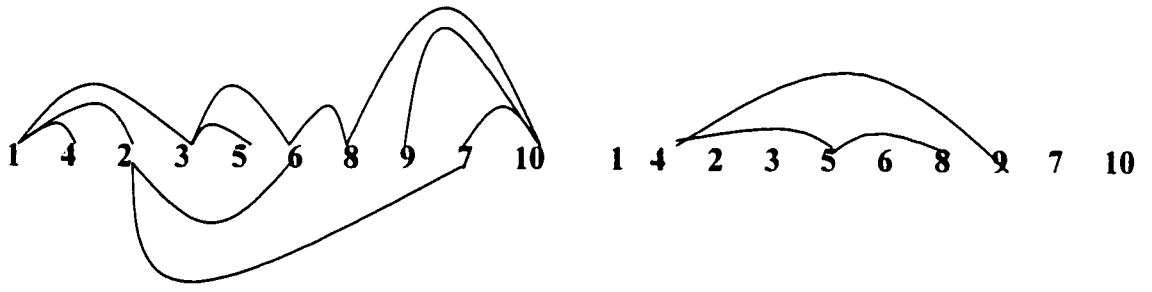


Fig. 1.4 Three page embedding of ordered set

This thesis however, does not view the vertices as an ordered set. In our case, for n vertices of a graph, all of $n!$ permutations can be considered.

1.3 Known Results

The pagenumber problem has been solved for several popular interconnection networks. The upper bound for pagenumber of planar graphs is 4 [Y]. Some planar graphs studied include square grids, X-trees, d-ary trees etc. Outerplanar graphs can be embedded in 1 page [BK]. A boolean d-cube admits a $(d-1)$-page embedding [CLR2]. A complete graph with n vertices can be embedded in $n/2$ pages [CLR2]. The optimal book-embedding for the FFT, benes, and barrel-shifter networks is 3 [G]. Details about the pagenumbers of these graphs are given in chapter 2.

1.4 Motivation

- Past research papers on the pagenumber problem do not propose any algorithm that determines the pagenumber of an arbitrary graph.

- Embedding a graph in a two dimensional layout has not been considered so far.

1.5 Contributions

- A review of the pagenumber problem with respect to its origin, past contributions and implementation is presented (chapter 2).

- Two two-dimensional pagenumber models (square and rook) for embedding graphs are proposed (Chapter 3).

- Two solutions using Hill Climbing, and one solution using Genetic Algorithms are described for the one and two-dimensional models. A unified framework for all three models is presented, in which a solution comprises of two permutations (nodes and edges). The three models differ by criteria for edge intersections (Chapter 5).

- Experimental results for several graphs, with respect to the one and two-dimensional models are obtained (Chapter 6).

  This thesis is a joint research with my supervisor, Professor Ivan Stojmenovic, Professor Albert Zomaya (University of Western Australia), and Mark Russell, and will be submitted for publication. Undergraduate student, Mark Russel implemented an algorithm for finding graph bandwidth and its transformation into a pagenumber, as part of a CSI 4900 project. He also provided experimental results for the 1-D pagenumber model, using genetic algorithms.

2. LITERATURE REVIEW

2.1 Planar Graphs

The book embedding problem has been studied for a variety of graphs. This section presents an overview of some known results for general types of graphs, and some popular interconnection networks.

In 1975, Bernhart and Kainen [BK] stated that

- the pagenumber of a graph G is 0 if and only if G is a path;
- the pagenumber is less than or equal to one, if and only if G is outerplanar.
- the pagenumber is less than or equal to two if and only if G is a subgraph of a Hamiltonian planar graph.

A *planar graph* (figure 2.1 a) is a graph that can be drawn such that none of the edges intersect; that is, the edges only touch each other where they meet at vertices.

In *outerplanar graphs* (figure 2.1 b), the nodes may be placed on the circumference of the circle such that no chords (corresponding to edges) of the circle intersect.

If a graph G has a spanning cycle Z (that is, a closed path that passes through every vertex exactly once), then G is called a Hamiltonian graph and Z a Hamiltonian cycle. *Planar Hamiltonian graphs* (figure 2.1 c), are planar graphs with a Hamiltonian cycle. *Subhamiltonian planar graphs* are subgraphs of planar Hamiltonian graphs. Examples include square grids (figure 2.3) and X-trees (figure 2.5).

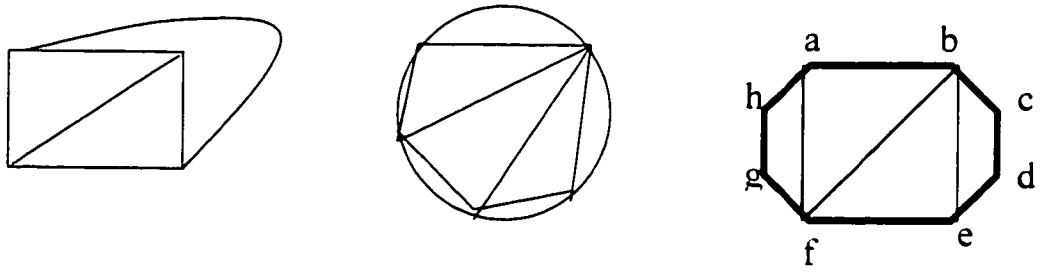


Fig. 2.1 a. Planar graph **b.** Outerplanar graph **c.** Planar Hamiltonian graph- Hamiltonian cycle is a-b-c-d-e-f-g-h-a

Theorem [BK]: A graph can be embedded in a one-page book if and only if, it is outerplanar.

Proof: If G is outerplanar, and is laid out on a circle, then cutting the circle between any two vertices and opening it out to form a line yields a one-page embedding of G [CLR2]. Conversely, given a one-page embedding of G, passing a line through the vertices of G in their order in the embedding, and joining the ends of the line together to form a circle, demonstrates G's outerplanarity. ♦

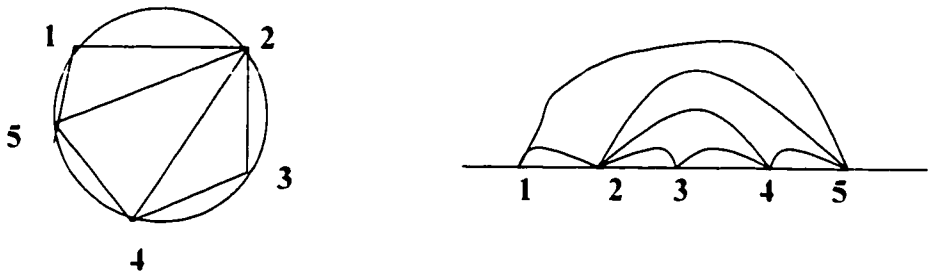


Fig. 2.2 Outerplanar graph and its one page embedding

As there exist planar graphs that are not Hamiltonian, it implies that there are planar graphs that require at least three pages. [BK] suggested that planar graphs have unbounded pagenumber. Buss and Shor [BS] then developed an algorithm to embed all planar graphs in 9 pages. Heath [H] reduced the number to 7. Istrail [I] found a six-page algorithm, followed by Yannakakis[Y] who showed that all planar graphs have an upper and lower bound of four pages.

Theorem [BK]: A graph G admits a 2-page embedding if, and only if, it is subhamiltonian, ie, a subgraph of a planar Hamiltonian graph.

Proof : A graph is subhamiltonian if it is embeddable in the plane so that its vertices lie on a circle, and each of its edges lies either totally within the circle or totally outside it. No edges cross in the layout. Given such a circular embedding of a subhamiltonian graph G, cutting the circle between any two of G's vertices yields a planar embedding of G in a line, with each edge lying either totally above the line (on page 1), or totally below it (on page 2). Conversely, given a 2-page embedding of the graph G, we view this embedding as placing G in a line with each edge lying totally above the line (page 1) or totally below it (page 2), and with no edges crossing. Pasting together the ends of the line containing G's vertices yields a circular embedding of G that witnesses G's subhamiltonian planarity. ♦

Square Grids (figure 2.3) comprise of a matrix of (nxn) nodes. They are planar and subhamiltonian.

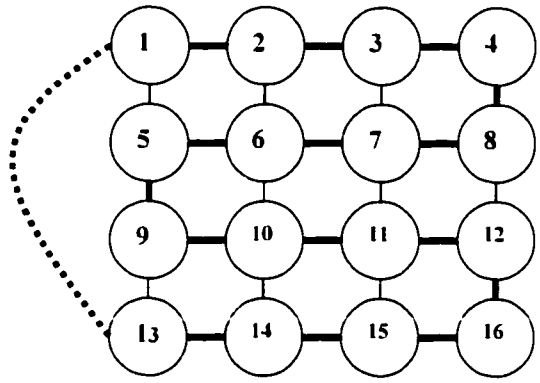


Fig 2.3 Square grid and Hamiltonian cycle formed by the grid

Since square grids are subhamiltonian, we know from theorem 2 that they can be embedded in two pages. The augmented hamiltonian path formed by row-by row alternated east-to-west and west-to-east sweeps leads to a two page embedding [CLR2]. The figure below shows a 4x4 grid, with the bold lines representing the hamiltonian cycle. All bold edges can be placed on page 1. As we traverse the hamiltonian cycle (obtained by adding an edge, e.g. dashed edge (1,13) in fig. 2.3) from node 1 to node 13, along the Hamiltonian path (marked in bold), all edges to the right of the nodes can be placed on page 2, while all edges to the left of the nodes can be placed on page 1. The two page embedding of the above grid is shown in figure 2.4.

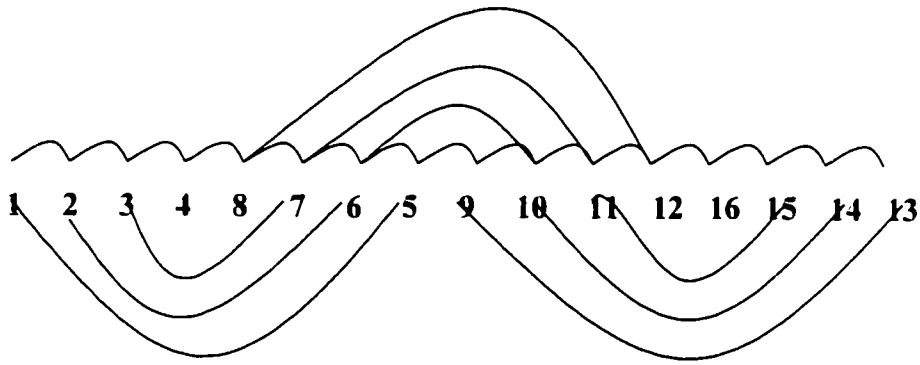


Fig. 2.4 Two page embedding of square grid

X-trees are defined as follows: The depth-d X-tree $X(d)$ (figure 2.5) is the edge augmentation of the depth-d complete binary tree that adds edges going across each level of the tree in left-to-right order. X-trees are planar and subhamiltonian.

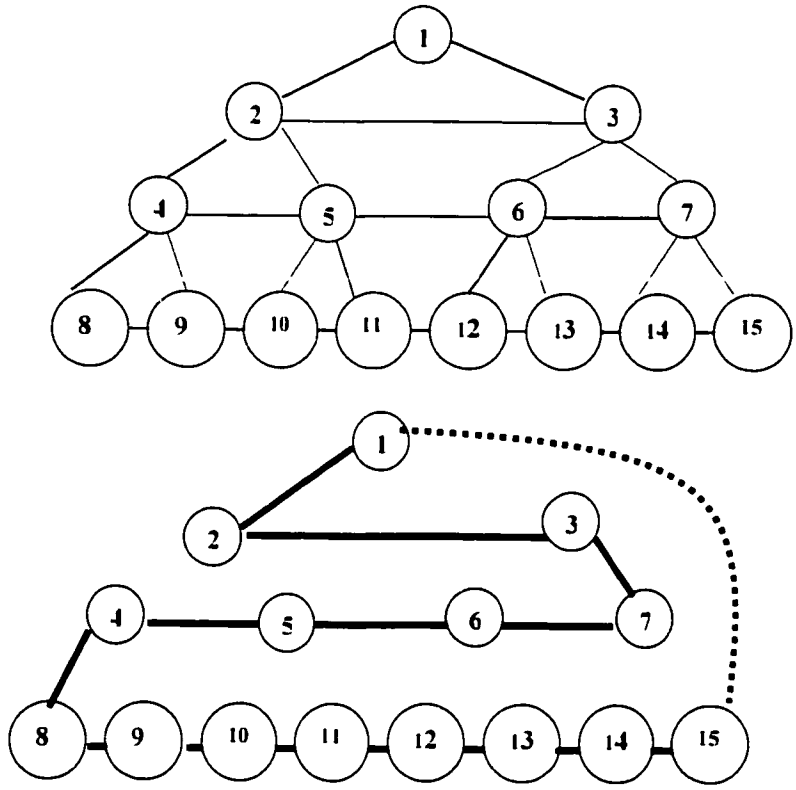


Fig. 2.5 X-tree and Hamiltonian cycle formed by the X-tree

Thus they admit a two-page embedding (theorem 2). The two page embedding is obtained from the cycle that runs across levels in alternating orders [CLR2]. The 2-page embedding of the edge-augmented X-tree in figure 2.5 is shown in figure 2.6. We start from node 1 and traverse the tree in the following order: 1-2-3-7-6-5-4-8-9-10-11-12-13-14-15. The bold edges are placed on page one, and while we traverse the path, all edges to the right of the path are placed on page 1, while all edges on the left of the path are placed on page two.

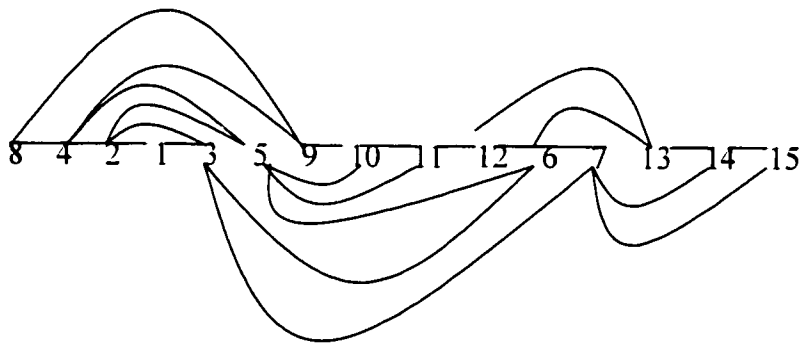


Fig. 2.6 Two page embedding of X-tree

A *d-ary tree* is a tree in which each node has at most d successors or child nodes. Every n-vertex d-ary tree can be embedded in a book having one page [CLR2]. The tree is laid out in preorder (root-left branch-right branch), and the edges are connected from left to right (Diogenes approach) in order to yield a one-page book embedding. Figure 2.7 shows a binary tree and its one page embedding.

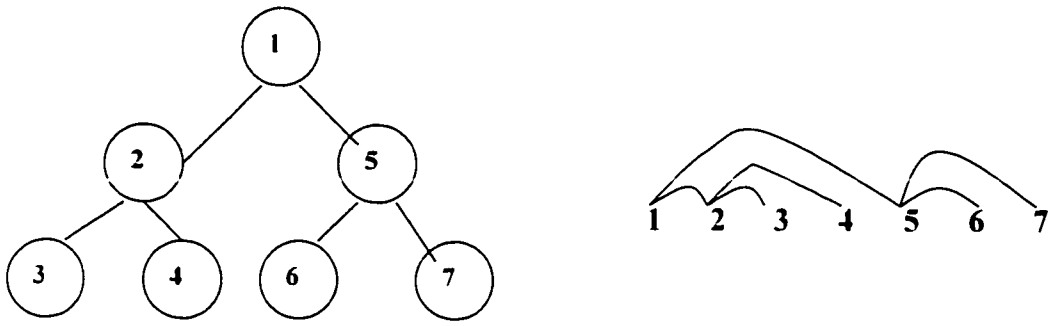


Fig. 2.7 Binary tree and its one page embedding

2.2 Hypercubes

A *boolean d-cube C(d)* has as vertices the set of all binary strings of length d. The edges of C(d) connect string-vertices x and y just when x and y are unit Hamming distance apart; that is, when there exist binary strings α,β of collective length n-1, such that $\{x,y\}=\{\alpha 0\beta, \alpha 1\beta\}$. For a dimension d, the number of nodes is $n=2^d$.

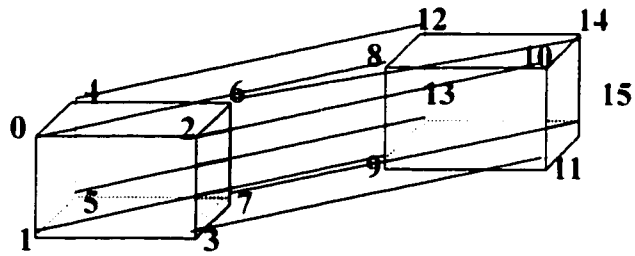


Fig. 2.8 4-dimensional hypercube

The boolean d-cube C (d) for $d \geq 2$, admits a (d-1)-page embedding. The efficient hypercube layout [CLR2] comprises of processors ordered along the edge of the book in Gray code order.

For a one-dimensional cube (d=1), the Gray code order is (0 1). For d=2, the Gray code order is obtained by making a mirror image of the Gray code order of the one-dimensional cube, and adding a "0" to the left of the digits in the left mirror image, and "1" to the left of the digits in the right mirror image. The mirror image for d=1 is (0 1 1 0). Now we add "0" to the left of 0 and 1 (the first two digits), in order to obtain 00 and 01. Next, we add a "1" to the left of 1 and 0 in order to obtain 11 and 10. Hence, for d=2,

the gray code order is (00 01 11 10). Similarly, for d=3, the Gray code order is (000 001 011 010 110 111 101 100).

The figure 2.9 shows the 3-page layout for the 4-dimensional hypercube of fig.2.8. Each 3-D cube (with 8 nodes) can be embedded in two pages. The links (edges) that connect the two 3-D cubes to form a 4-D cube, are embedded in the third page (represented by the dashed lines in figure 2.9). Thus a 4-D cube can be embedded in 3 pages. Similarly, a d-dimension cube can be embedded in d-1 pages. For each new dimension, one additional page is needed. Two copies of (d-1)-dimensional hypercube may use the same pages, with second copy placed as a mirror image of the first copy. All additional edges between the two copies, which complete the d-dimension hypercube can be placed on one additional page.

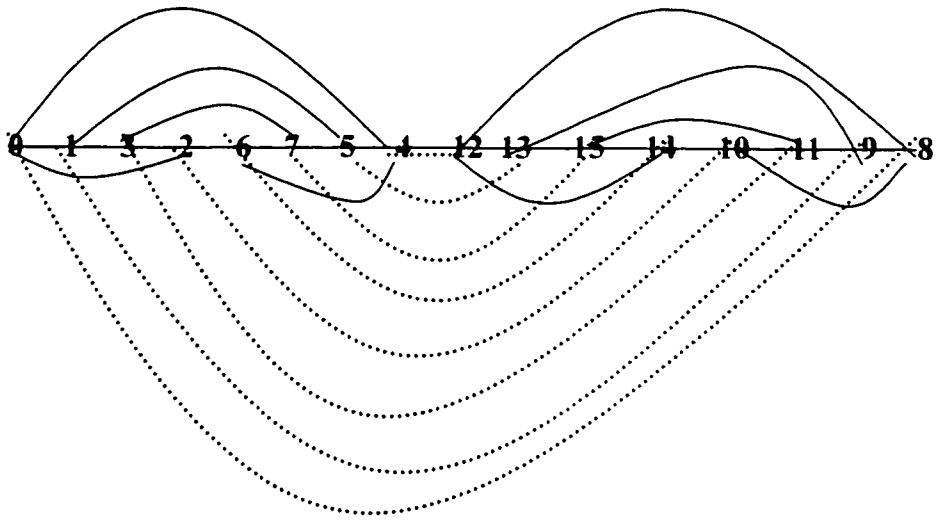


Fig 2.9 Gray code layout for 4-dimensional hypercube.

2.3 Complete Graphs

A *complete graph* (figure 2.10) is a graph in which every pair of vertices is adjacent. A complete graph with n nodes is referred to as K_n.

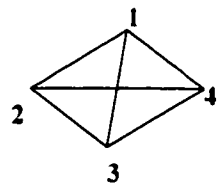


Fig. 2.10 Complete graph with 4 vertices

We can show that this graph can be embedded in n-1 pages by stating that for each vertex i, $1 \leq i \leq n$-1, all edges incident to i can be placed on the i-th page. However, fewer pages may suffice.

The complete graph K_n is embeddable in $n/2$ pages [CLR2]. To show that an $n/2$ page embedding is possible, consider the following way to layout K_n. Place the vertices $0,1,\ldots n\text{-}1$ evenly spaced on a circle. For each vertex v, $0 \le v < n/2$, draw the line-graph L_v as indicated in the following illustration, in which all arithmetic is modulo n and in which double dashes (==) denote edges of the line:

$v == v+1==v-1==v+2==v-2==\ldots==v+(n/2)-1==v-(n/2)+1==v+(n/2)$.

Clearly, each such line is composed of non-crossing chords of the circle; hence, it is embeddable on a single page. Next, every edge of K_n appears in precisely one line (no edge is 'forgotten'). To verify this, note that each vertex w is an endpoint of (hence has valence 1) precisely one line in the circle, namely, $L_{w \bmod n/2}$, and has valence 2 in all other lines, so that in all, we have $2((n/2)-1)+1 = n-1$ edges that leave w. Thus all edges from each node are placed on a page. Finally, no two lines have any common edge. Hence, if we snip the circle between any two vertices, thereby laying K_n out in a line, and if we color the edges of K_n according to which line L_v they line in, then we obtain an embedding of K_n in a $n/2$-page book. As an example, consider the complete graph shown in figure 2.11.

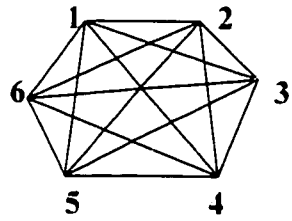


Fig 2.11 Complete graph with n=6

The non-crossing chords of the circle and a three page embedding of the above graph are shown in figure 2.12.

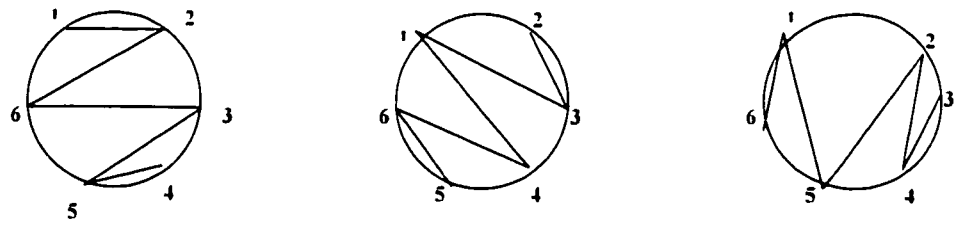


Fig. 2.12 non-crossing chords of the circle

2.4 Pagenumbers of other graphs

A *complete bipartite graph* can be partitioned into two subsets of nodes such that each node is joined to every node in the other subset.
The *fast fourier transform graph* (FFT) F (N) is defined as follows: Let N be a power of 2. The N-input FFT network F(N) is defined inductively as follows:

- F(2) is the complete bipartite graph $K_{2,2}$ on two input vertices $i_{2,1}$, $i_{2,2}$ and two output vertices $o_{2,1}$, $o_{2,2}$.
- F(N), N>2, is obtained from two copies of F(N/2) and N new input vertices $i_{N,1}$, $i_{N,2}$,....$i_{N,N}$. For each a $\in$ {1,2,....N/2}, edges creating one copy of $K_{2,2}$ with "inputs" $i_{N,a}$ and $i_{N,a+(N/2)}$ and "outputs" $i_{N/2,a}$ and $i'_{N/2,a}$ are added (primed vertices come from the second copy of F(N/2)).

The *Benes graph B(n)* is defined like the FFT network, along with the addition of N new output vertices $o_{N,1}$, $o_{N,2}$,.... $o_{N,N}$ at each step For each $a \in$ {1,2,....N/2}, edges creating one copy of $K_{2,2}$ with "inputs" $o_{N/2,a}$ and $o'_{N/2,a}$ and "outputs" $o_{N,a}$ and $o_{N,a+(N/2)}$ are added.

The *barrel shifter graph* is defined as follows: For N a power of 2, the N-input (log N)-stage barrel shifter S(N) has vertices {(a, b):0 $\le a \le$ log N, 0$\le b<N$}, and edges that connect vertex (a, b), 0 $\le a \le$ log N, 0 $\le b<N$ to vertex (a+1,b) and to vertex (a+1,b+2^a (mod N)). The input vertices have a=0 and the output vertices have a =log N.

Games[G] proposed an optimal 3-page book embedding for the FFT, Benes, and the barrel-shifter networks. Since all three networks are nonplanar, they require three pages [BK].

The vertices of the *deBruijn graph* [SI] *D(m,k)* are all (k,m)-variations, that is, all variations $d_1d_2...d_k$ such that $d_i \in$(0,1,....m-1) for 1$\le i \le k$. Thus the number of nodes is n= m^k. Two variations (nodes) are connected by an edge if and only if they are obtained from each other by a shift for one position to the left or right and appending an arbitrary element to the position that becomes empty by shifting.

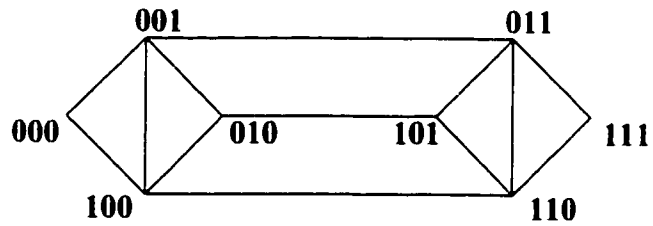


Fig. 2.13 deBruijn graph

The deBruijn graph can also be defined as follows [dB] : The order-n deBruijn graph

n $n-1$
D(n) has node - set Z_2 ; given $y \in Z_2$, two arcs are incident out of each node βy: the shuffle arc that leads to $y\beta$ and the shuffle-exchange arc that leads to $y\beta'$.

The order-*n shuffle-exchange graph S(n)* has node-set Z ; given $y \in Z$ and $\beta \in Z2$, the shuffle arc leads node βy to node $y\beta$ and the exchange arc leads node $y\beta$ to node $y\beta'$.The shuffle-exchange graph *S(n)* is a subgraph of deBruijn graph.

Obrenic [O], presented a construction for embedding deBruijn and shuffle-exchange graphs in five pages. No efficient book-embeddings have been found so far for shuffle-like networks (deBruijn and shuffle-exchange graphs). The very weak upper bound for the much broader class of bounded-degree graphs applies, but is non-constructive, so it follows from [CLR2] that there exist book-embeddings of N-node deBruijn (shuffle-exchange) graphs with pagenumber $O(\sqrt{n})$.The best known lower bound on the pagenumber of deBruijn and shuffle-exchange graphs remains 3, which follows from the graphs' nonplanarity.

Heath and Istrail [HI] proved that genus g graphs could be embedded in O (g) pages. A lower bound Ω(sqrt (g)) was also derived.

A *k-tree* is defined inductively in the following way: -

- The k-vertex complete graph K_k is a k-tree.
- If G is a k-tree, then adding a new vertex to G that is adjacent to k vertices that induce a K_k in G results in a k-tree. Figure 2.14 shows a 2-tree.

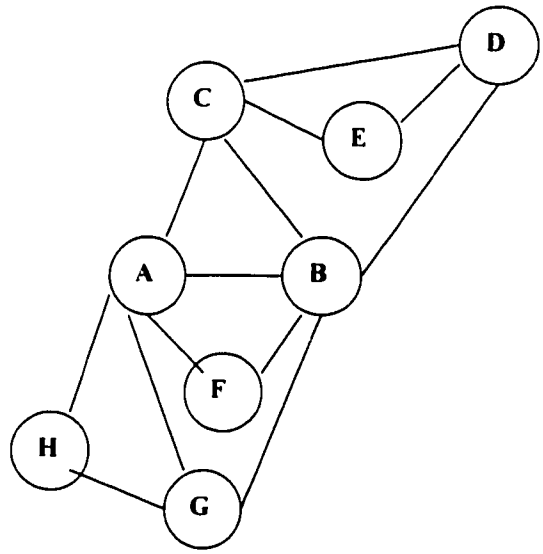


Fig. 2.13 2-tree

Recently, Ganley and Heath [GH] proved that the pagenumber of a k-tree is at most k+1, and that there exist k-trees that require k pages. The 2-tree in figure 2.13 has a pagenumber of at most 3.

2.5 Bandwidth and Pagenumber

Let Φ be the permutation (layout) of nodes of a graph G. The bandwidth bw (G,Φ) of G is bw $(G, \Phi)= \max\{ | \Phi(u)-\Phi(v)| (u, v) \in G\}$. In words, the bandwidth is the length of the longest edge in the layout. Figure 2.14 shows a graph and its corresponding embedding. The longest edge is (1,4), and hence the bandwidth for the above graph is 3 (for the layout $\Phi = (1234)$).

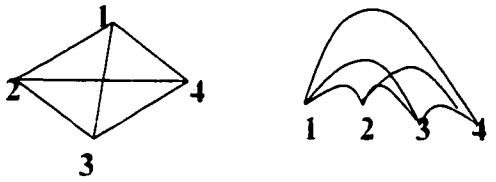


Fig. 2.14 Graph and its embedding

The bandwidth bw (G) of G is $bw(G)= \min bw$ (G, Φ). In words, the bandwidth of G is the smallest bandwidth of any of its layouts.

A graph with bandwidth k is called a bandwidth-k graph. The maximal bandwidth–k graph G on n vertices is the graph with vertex set $v=\{ 1,2,\ldots n\}$ and edge set $E =\{(i,j)| 1 \le |i-j| \le k\}$. Any bandwidth-$k$ graph having n vertices is a subgraph of G.

Swaminathan, Giriraj, and Bhatia [SGB] showed that bandwidth-k graphs could be embedded in k-1 pages with respect to a linear ordering of its vertices. The ordering of vertices is not the same one used to define the bandwidth. We have simplified the algorithm [SGB] that embeds bandwidth-k graph into k-1 pages.

The pagenumber solutions obtained after determining the bandwidth of graphs may be used as initial solutions in our evolutionary computing algorithms for computing pagenumber. The bandwidth of an arbitrary graph can also be found by applying evolutionary computing techniques.

The following algorithm gives a (k-1)-page embedding for one particular layout of a bandwidth-k graph:

Algorithm: (k-1)-page embedding.

Input: the maximal bandwidth-k graph, G on n vertices for $k \geq 2$ and $n \geq k$.
Output: a layout ϕ of G and a (k-1)-page embedding of (G, ϕ).

```
Begin
    for i : = 1 to n do
    begin
       if (i mod 2 =1)
          then φ(i) : = (i div 2) + 1;
          else φ(i): = n –(i div 2) + 1;
    end;

   for i : = 1 to n do
   begin
      j : = ( (i-1)mod (k-1) ) +1
      for p: = i+1 to i+k do
         if p < = n
           then place edge (i,p) on page j
  end
End.
```

2.6 Related Layouts

We have discussed only graphs and their layouts with respect to the book model. There exist other VLSI layout models for graphs. Yeh and Parhami [YP] have studied other layouts for complete and star graphs. One such layout is the grid model [T]. The network is viewed as a graph whose nodes correspond to processing elements and edges to wires. The graph is then embedded in a 2D grid, where wires have unit width and a node of degree d occupies a square of side d. Wires can run either horizontally or vertically along grid lines. The area of a layout is the area of smallest rectangle that contains all nodes and wires. Another layout for complete graphs, called the colinear layout, makes use of tracks to place links between nodes.

To obtain a 1-dimensional colinear layout, the nodes, labeled 1 through n are placed along a row. A link is of type-i if it connects two nodes whose addresses differ by i. The $n(n-1)/2$ links in a complete graph, K_n, can be classified into types 1,2...n-1, and there are n-i type-i links. To embed the edges of K_n, all type-1 links are placed in one track, and all type-2 links in two tracks, where links connecting nodes with odd addresses are put in one track and links connecting nodes with even addresses in the other. All type-i links are placed in min(i, n-i) tracks, for i = 3,4,...n-1. In other words, type-i links are placed in i tracks if $i \leq n/2$, where links are put in the same track if the remainders of their node-addresses modulo i are the same, and each of the n-i type-i links is placed in a different track if $i > n/2$. Figure 2.15 shows an embedding for a K_4 graph in four tracks.

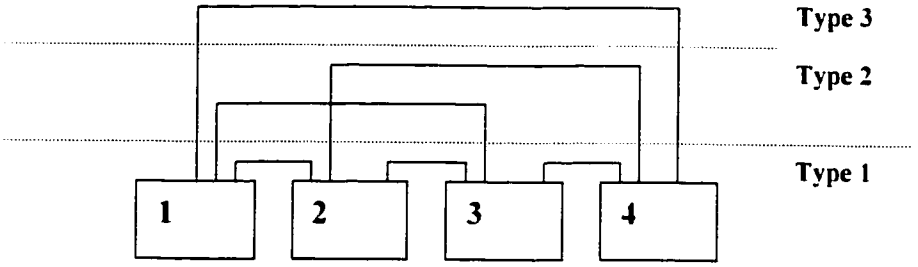


Fig. 2.15 Embedding for a complete graph with four vertices

3. 2-D PAGENUMBER PROBLEM

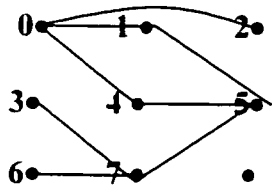
We introduce a new 2-D model for embedding graphs. The nodes of the graph are placed anywhere on the plane. Edges of the graph are then added by joining corresponding nodes in an arbitrary way. Each edge is placed on a page, such that each page is a planar graph. The problem is to determine the minimum number of pages needed to embed the graph. As it is difficult to design any general algorithm with this unrestricted algorithm, we restrict to several models for which we propose evolutionary computing solutions.

3.1 Square Model

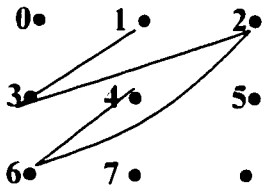
In order to design an algorithm for an arbitrary graph, it is necessary to include some restrictions on nodes and edges. The first layout model that is considered is the 2-D square model, where nodes are arranged so as to form a square grid. A graph with n nodes is embedded in a grid with $[\sqrt{n}]$ rows and $[\sqrt{n}]$ columns. Each node is placed on a separate grid point. Node placement is specified by their permutation. Nodes are placed in row major order, without skipping any position. For a given graph G (V, E), we introduce two methods for embedding its edges.

Straight-line method: In this method, edges are embedded by joining nodes with straight-line segments. However, if a straight line passes through another node, a line that avoids these nodes replaces it.

As an example, consider the 2-D embedding of a cube, in two pages. Nodes are placed on a grid, and the edges are embedded as indicated.



Page 1: Embedded edges are (0,1), (0,2), (0,4), (1,5), (3,7), (4,5), (5,7), (6,7)



Page 2: Embedded edges are (1,3), (2,3), (2,6), (4,6)

Fig 3.1 2 –page embedding of cube

Horizontal-vertical method: In this method, edges are embedded by using horizontal and vertical line segments only. However, in most cases there will be intermediate nodes that should be avoided. Thus they are replaced by nearby lines on one of two possible sides. For example, edge (1,6) in figure 3.2 consists of a horizontal segment 0-1 and a vertical segment 0-3-6, and is replaced by a dashed line outside (alternative inside line may also be used).

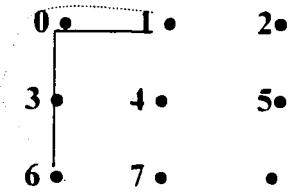
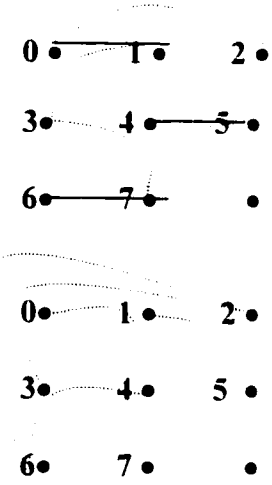


Fig. 3.2 Horizontal-vertical method

As an example, consider the 2-D embedding of a cube, in two pages.



Page 1: Embedded edges are (0,1), (0,2),(1,3),(3,7),(4,5),(5,7),(6,7)

Page 1: Embedded edges are (0,4),(1,5)(2,3),(2,6),(4,6)

Fig. 3.3 Two page embedding of a cube

Figure 3.4 shows a two-dimensional embedding of a four-dimensional hypercube, which is an example for both the straight-line and horizontal-vertical methods. The square model with respect to the straight-line method is described further in sections 3.4 and 3.5.

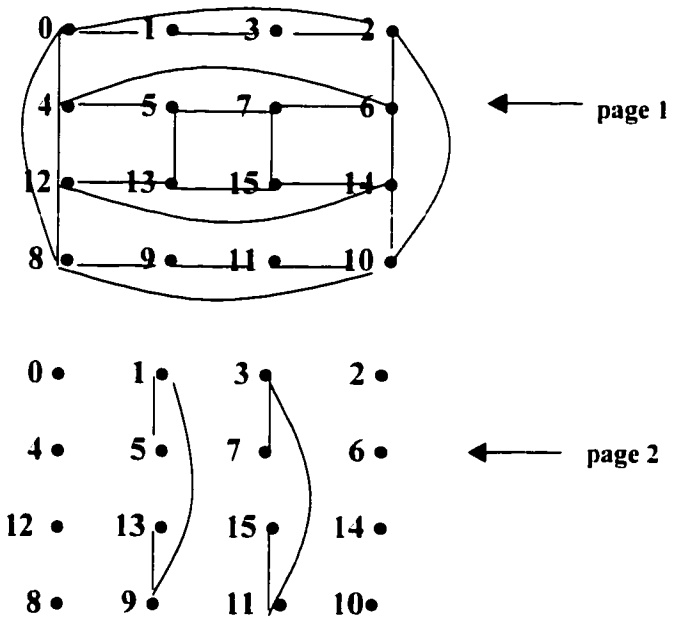


Fig. 3.4 2-D embedding of a 4-D hypercube

3.2 Rook Model

Suppose that n nodes are placed in a plane. If any two nodes are on the same horizontal or vertical line, slightly rotate the plane so that this is no longer the case. This process does not affect the pagenumber. Thus we may assume that nodes are located on separate vertical and separate horizontal lines. We can further transform the embedding by translating nodes one by one, into discrete horizontal or vertical line positions, respecting their mutual order in both directions. This transformation also does not affect pagenumber, since non-intersecting edges will be transformed into corresponding non-intersecting edges (if straight method is applied). Thus position of nodes can be specified by two permutations, one for the horizontal, and the other for the vertical direction. For example, figure 3.4 can be characterized by two permutations 62734051 and 02134657.

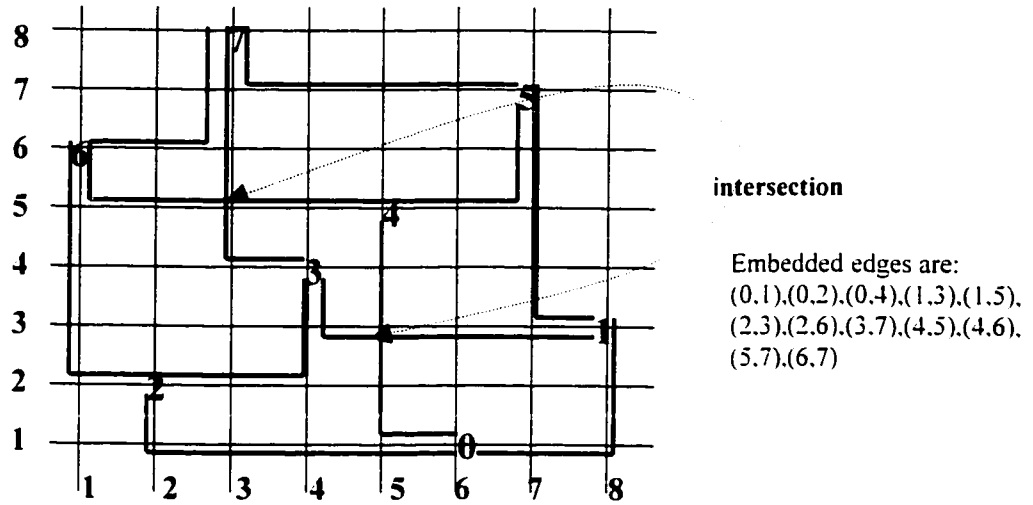


Fig. 3.4 Embedding of a cube with two edge intersections

Edges can be inserted using the straight-line method or horizontal-vertical method. In both cases, in order to present a general algorithm, we decided to restrict the ways edges are embedded. For instance, figure 3.4 shows a cube embedding, with edges drawn horizontally and then vertically (horizontal-vertical method). We assume that edges are recorded as (a, b) with $a<b$. For example, edge (3,7) in figure 3.4 is drawn horizontally, and then vertically. Edges can also be connected by drawing the vertical line first, and then the horizontal line. However, we have chosen the former method.

There are two edge intersections in figure 3.4, and thus two pages may be needed. Vertical or horizontal line segments having a common node, n, do not intersect, as each can be placed on a different track that originates from n. For example, the edges (1,5) and (4,5) have a common node, 5. The vertical line connecting (1,5), and the vertical line connecting (4,5) are placed on different tracks. Figure 3.5 shows a node n which is common to three edges (a,n), (b,n), (c,n). The bold lines show the line segments on different tracks.

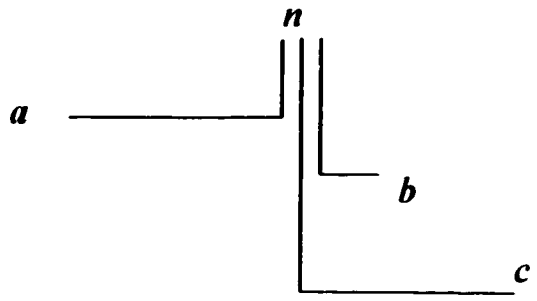


Fig. 3.5 three non-intersecting line segments

3.3 Edge intersections in 1-D

In both the one and two-dimensional models, intersection of edges is determined based on certain criteria. Let us first consider the 1-D pagenumber model. We can determine if two edges *ab* and *cd* intersect, based on the positions *a'*, *b'*, *c'*, and *d'*, of *a*, *b*, *c*, and *d*, respectively, in the permutation of nodes, *P*. The positions can be treated as x-coordinates of nodes.

Fig. 3.6 intersection of two edges

The edge *ab=(3,1)* intersects with edge *cd=(2,4)* (figure 3.6) because *c'* =2 (which is the position of *c* in the permutation P) is positioned in between *a'* =1 and *b'*=4, and *d'*=5 is positioned after *b'*. The permutation containing the positions *a' b' c' d'* is called the inverse permutation of *P*, and is used to determine whether or not edges *ab* and *cd* intersect.

The function *Intersect-1-D* is used to check if an edge *a'b'* crosses with an edge *c'd'*, where *a'*, *b'*, *c'*, *d'* are positions (or, alternatively, x-coordinates) of end points.

```
Function Intersect-1-D (a', b', c', d' : integer ) : boolean;
Begin
    If a' < b'
      then
          begin
             min-ab : = a' ; max-ab: = b'
          end;
     else
          begin
             min-ab: = b' ; max-ab : = a'
          end;
   If c' < d'
     then
         begin
            min-cd : = c' ; max-cd : = d'
         end;
     else
        begin
           min-cd : = d' ; max-cd : = c'
        end;
  Intersect-1-D : = false;
  If min-ab < min-cd AND  min-cd < max-ab AND max-ab < max-cd
    then  Intersect-1-D : = true;
  If min-cd < min-ab AND min-ab < max-cd AND max-cd < max-ab
```

then *Intersect-1-D* : = true;
End.

An inverse permutation, *IP* stores the positions of each node in *P*. *IP* is formed by placing at each position *i*, $1 \leq i \leq n$ (*n* corresponds to the maximum number in the permutation) the position of element *i* in *P*. The algorithm for computing *IP* of a permutation *P* is described below.

Procedure Inverse-Permutation (*P*: **array**[1..*n*] **of integer** ; **var** *IP*: **array**[1..*n*] **of integer**);
{Input: an array P which consists of a permutation of vertices.
Output: the inverse permutation IP of P.}
Begin
For *i*: = 1 **to** *n* **do**
IP [P[i]] : = *i*;
End.

For example, if the permutation *P* = (3 2 5 1 4), then the first element of *IP* is 4 because the number 1 is in position 4, in *P*. Similarly, the number 2 is in position 2 (thus the second element of *IP* is 2), number 3 is in position 1, number 4 is in position 5, and number 5 is in position 3. Hence, *IP* = (4 2 1 5 3).

The function *Intersect-1-Dim* is used to check if for a graph, G(V,E), edge ab intersects with edge cd.

Function *Intersect-1-Dim* (*a,b,c,d*: **integer** ; *P*: **array**[1..*n*] **of integer**) :**boolean;**
Begin
Inverse-Permutation (P, IP);
Intersect-1-Dim: = Intersect-1-D (*IP[a], IP[b], IP[c], IP[d]*)
End.

3.4 Edge Intersections in 2-D

This section discusses the straight-line method for edge embedding in the 2-D model. We make use of equations of lines in order to determine whether or not two line segments (edges) intersect. Consider the two line segments shown below.

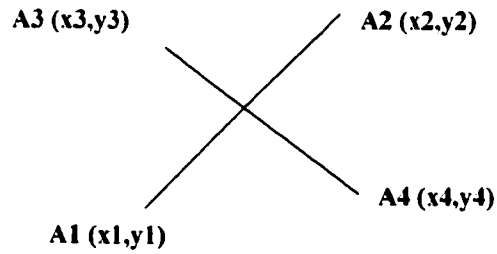


Fig 3.7 Intersection of two lines

Two vectors, $P1$ and $P2$ give the direction of edges. $P1 = A2\text{-}A1 = (x2\text{-}x1, y2\text{-}y1)$ and $P2 = A4\text{-}A3 = (x4\text{-}x3, y4\text{-}y3)$.
The equations for the two lines are $X = A1 + uP1$ and $X = A3 + vP2$

The two lines intersect if the following condition holds: $A1 + uP1 = A3 + v\,P2$

Substituting for $A1$, $A3$, $P1$ and $P2$, we have

$(x1,y1) + u(x2\text{-}x1, y2\text{-}y1) = (x3,y3) + v(x4\text{-}x3, y4\text{-}y3)$

Therefore,
$x1 + u(x2\text{-}x1) = x3 + v(x4\text{-}x3)$, and
$y1 + u(y2\text{-}y1) = y3 + v(y4\text{-}y3)$.

These equations can be represented as $au + bv = c$ and $du + ev = f$, where,
$a = x2\text{-}x1$; $b = x3\text{-}x4$; $c = x3\text{-}x1$; $d = y2\text{-}y1$; $e = y3\text{-}y4$; $f = y3\text{-}y1$.

Solving the simultaneous equations, we get $u = (ce\text{-}bf) / (ae\text{-}bd)$ and $v = (af\text{-}cd) / (ae\text{-}bd)$

The conditions for the two line segments to intersect are given by $0 < u < 1$ and $0 < v < 1$.
If either of these two conditions is not true, then the line segments do not intersect.

Special Cases

Case 1: The two lines are parallel if $(ae\text{-}bd)=0$ and $(af\text{-}cd) \neq 0$. Hence, the lines do not intersect.

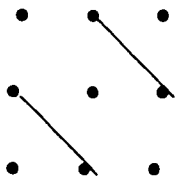


Fig. 3.8 parallel lines

Case 2: The two lines overlap if $(ae\text{-}bd)=0$ and $(af\text{-}cd) = 0$.

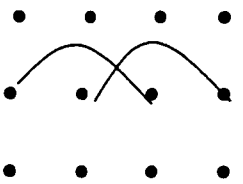


Fig. 3.9 overlapping lines

The function *Intersect-1-D (x1,x2,x3,x4)* is called when y1=y2=y3=y4, or when lines overlap on a diagonal.
The function *Intersect-1-D (y1,y2,y3,y4)* is called when x1=x2=x3=x4.

Case 3: This case occurs when $u = 0$ or 1 or $v = 0$ or 1. This case may be divided into two cases:

(a) End point of one edge lies on other edge (but not on the end point of the other edge).

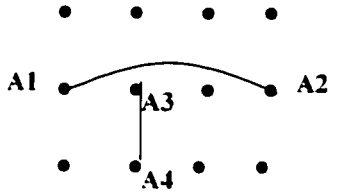


Fig. 3.10 end point on another edge

(b) End point of one edge coincides with end point of other. In this case, there is no intersection.

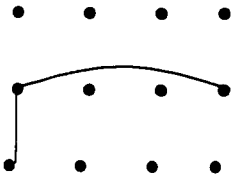


Fig. 3.11 two edges with common end point

Let us look at case 3(a) in more detail. If an edge passes through several intermediate nodes, it is replaced by an edge that avoids these nodes. We will refer to such an edge as a *curved edge*. If edge *A1A2* is a curved edge, then it will be 'skewed' at the same side as the normal to *A1A2*. Such a decision is taken to reduce the search space in algorithms that search for the pagenumber. Figure 3.12 shows a curved edge *A1A2*, skewed in the same side as the normal to it.

Given two end points of an edge, $A1(x1,y1)$ and $A2(x2,y2)$, we define, vector $\vec{p} = (x2-x1, y2-y1)$, and vector $\vec{n} = (y1-y2, x2-x1)$, which is normal vector to $\vec{p}$. That is, the dot product $\vec{n}.\vec{p} = 0$.

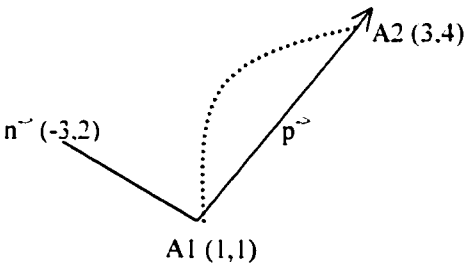


Fig. 3.12 edge A1A2 and its normal

The normal will always be to the left of an edge *A1A2*. If an end point *A3* or *A4* (say *A3*) of another edge lies on the line *A1A2* (in which case u=0 or 1, or v=0 or 1), then if *A4* and the normal to *A1A2* are on the same side, the edges will intersect. On the other hand, if they are on opposite sides, the edges will not intersect. These situations are depicted by figure 3.13a and 3.13b.

Fig. 3.13 edge intersection when end point of one edge lies on other edge

Given two lines, *A1A2* and *A3A4*, if n is the normal to *A1A2*, and point *A3* lies on *A1A2*, then,

- the dot product n . *A2A4* > 0 implies n and *A4* are on the same side
- the dot product n . *A2A4* < 0 implies n and *A4* are on opposite sides

where *A2A4* = (*x4-x2,y4-y2*) (*A1* may be used in the calculation, instead of *A2*).

This function is called *Same-Side*, and is defined as follows, for the case v = 0, in figure 3.13.

```
Function Same-Side (x3,y3,x4,y4,x1,y1,x2,y2: integer):boolean;
Begin
   {determine normal}
    nx: =y1-y2; ny: = x2-x1;

  {determine A2A4}
   vectorx: = x4-x2; vectory: = y4-y2;

   if (nx* vectorx + ny *vectory) > 0  {dot product }
      then Same-Side : = true;
      else Same-Side: = false;
End.
```

There are three more cases to be considered when the end point of an edge lies inside another edge. Figure 3.14 shows the three cases. where u=0. u=1. and v=1.

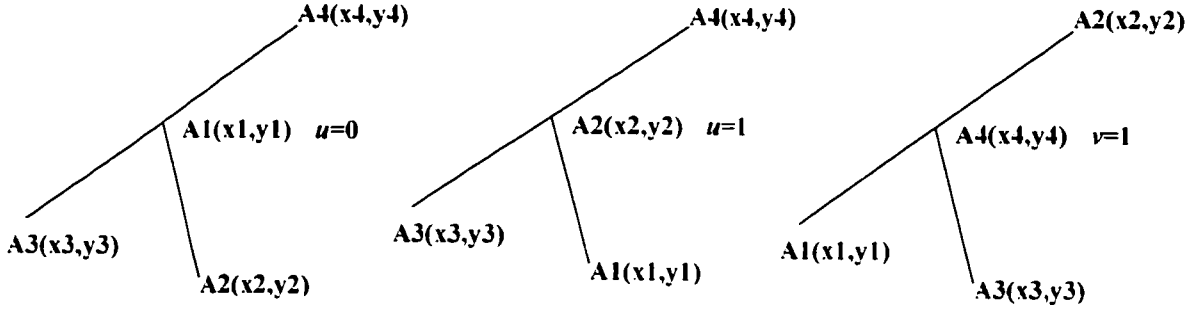


Fig. 3.14 cases u=0. u=1.v=1

The function *Same-Side(x1,y1,x2,y2,x3,y3,x4,y4)* is called to determine whether or not *A2* is on the same side as the normal to *A3A4* (for u=0). Similarly. the functions *Same-Side(x2,y2,x1,y1,x3,y3,x4,y4)* and *Same-Side(x4,y4,x3,y3,x1,y1,x2,y2)* are called for cases u=1 and v=1. respectively.

The function Int-Edges checks whether the edges *A1(x1,y1)*. *A2(x2,y2)* and *A3(x3,y3)*. *A4(x4,y4)* cross.

```
Function Int-Edges (x1,y1,x2,y2,x3,y3,x4,y4 : integer):boolean;
var
   a, b, c, d, e, f : integer; u,v : real;
begin
a: = x2-x1; b: = x3-x4; c: = x3-x1; d: = y2-y1; e: = y3-y4; f: = y3-y1;
if ae-bd = 0
  then if af-cd = 0 {overlapping lines}
         then if x1=x2
                then Int-Edges: = Intersect-1-D (y1,y2,y3,y4);
                else Int-Edges: = Intersect-1-D (x1,x2,x3,x4);
         else Int-Edges: = false; {parallel lines}
  else begin
       u: = (ce-bf)/(ae-bd); v: = (af-cd)/(ae-bd);
       if u<0 OR u>1 OR v<0 OR v>1 then Int-Edges: = false
       if u>0 AND u<1 AND v>0 AND v<1 then Int-Edges: = true;{intersecting lines}
       if (u=0 OR u=1) AND (v=0 OR v=1) then Int-Edges: = false; {common end point}
       if u=0 AND v>0 AND v<1 then Int-Edges: = Same-Side(x1,y1,x2,y2,x3,y3,x4,y4);
       if u=1 AND v>0 AND v<1 then Int-Edges: = Same-Side(x2,y2,x1,y1,x3,y3,x4,y4);
       if v=0 AND u>0 AND u<1 then Int-Edges: = Same-Side(x3,y3,x4,y4,x1,y1,x2,y2);
       if v=1 AND u>0 AND u<1 then Int-Edges: = Same-Side(x4,y4,x3,y3,x1,y1,x2,y2);
       end
end;
```

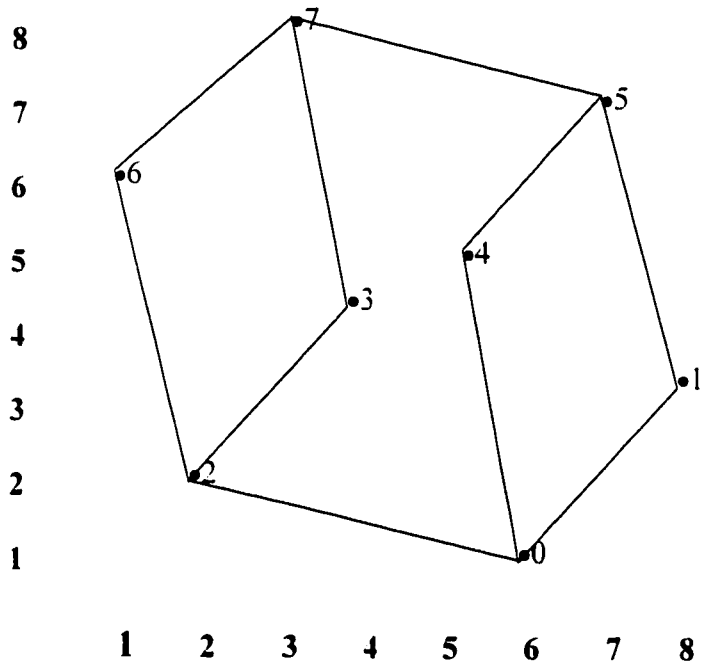
3.5 Edge Intersections in Rook Model

In this model, no two nodes can be placed in the same row or column. Edges can be embedded using either straight-line or horizontal-vertical method. Figure 3.4, given earlier, shows a 2-dimensional rook model with seven nodes. A grid is used to place all the nodes.

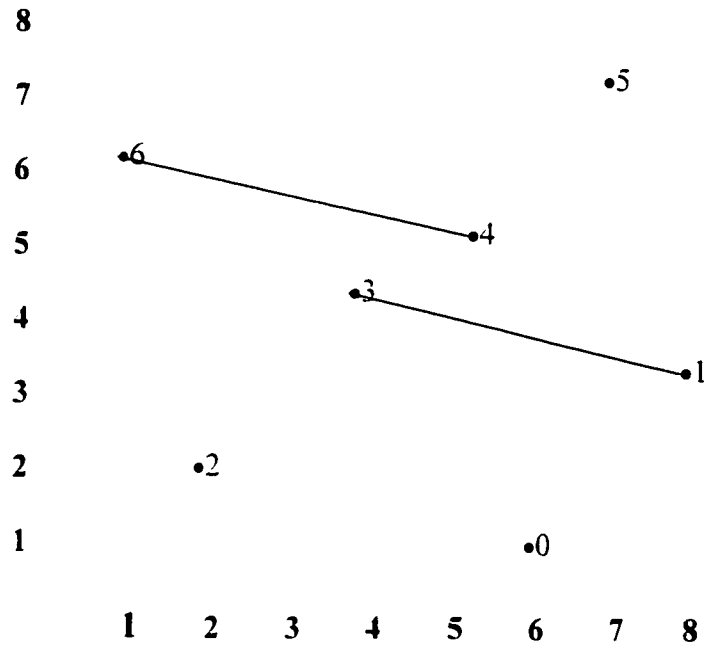
The solution space comprises of a permutation of nodes *Px*, in the *X* –axis, a permutation of nodes *Py*, in the *Y*-axis, and a permutation of edges. The table below shows each node and its location, which is represented by *Px* and *Py*. For example, the coordinates of node 1 are (8, 3).

NODES	0	1	2	3	4	5	6	7
Px	6	8	2	4	5	7	1	3
Py	1	3	2	4	5	7	6	8

Figure 3.15 shows a two-page edge embedding of a cube, with respect to the straight-line method.



Page 1: embedded edges are (0,1),(0,2),(0,4),(1,5),(2,3), (2,6),(3,7),(4,5),(5,7),(6,7)



Page 2: embedded edges are (1,3), (4,6)

Fig. 3.15 2-page embedding of cube with straight-line method

The function Straight-Line-Rook checks whether an edge A1A2 intersects with an edge A3A4, in the straight-line method of edge embedding. A1, A2, A3, A4 are four elements from a permutation of nodes.

Function Straight-Line-Rook (*A1,A2,A3,A4*: **integer**; *Px, Py* :**array**[1..*n*] **of integer**) :**boolean**

Begin

{obtain coordinates of edges A1A2 and A3A4}
x1: = *Px* [A1]; *y1*: = *Py* [A1];
x2: = *Px* [A2]; *y2*: = *Py* [A2];
x3: = *Px* [A3]; *y3*: = *Py* [A3];
x4: = *Px* [A4]; y4: = *Py* [A4];
Straight-Line-Rook : =Int-Edges (*x1,y1,x2,y2,x3,y3,x4,y4*);

End.

Figure 3.16 shows an edge (*A1A2*) intersecting with edge (*A3A4*), with respect to the horizontal-vertical method. In order to determine whether the edges intersect, we consider the coordinates of the vertical line *A4*, and the horizontal line *A1* (shown in bold). These correspond to the coordinates (*x4,y4*),(*x4,y3*) and (*x1,y1*),(*x2,y1*).

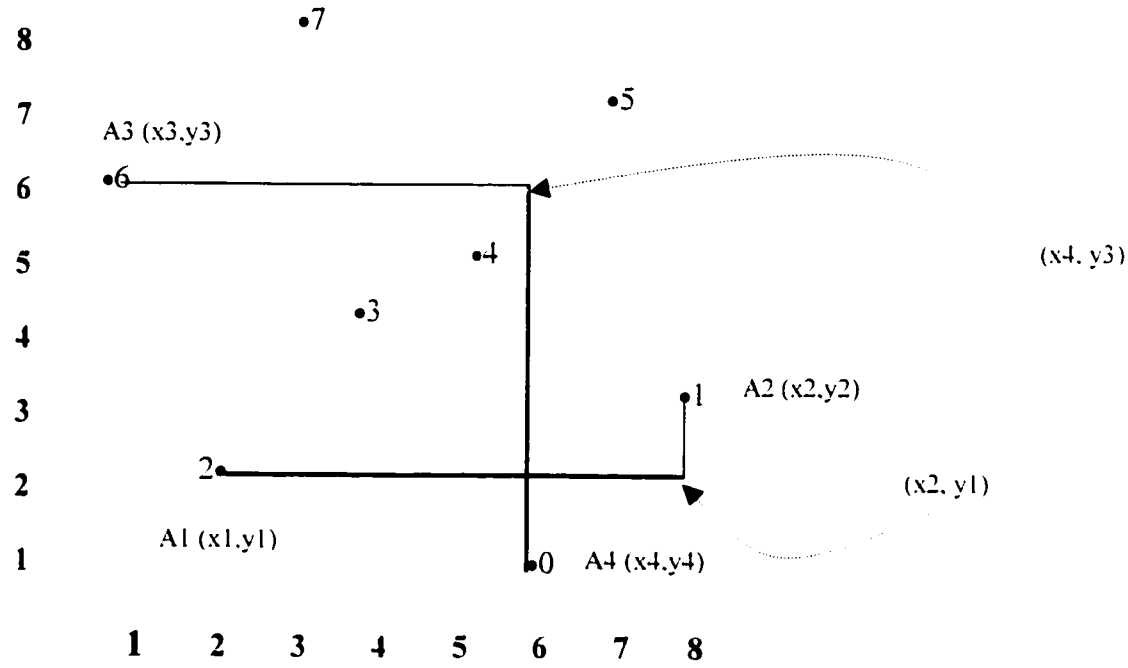


Fig. 3.16 horizontal line at A1 intersects with vertical line at A4

There are two cases to be considered here:

Case 1: Horizontal line of *A1A2* intersects with vertical line of *A3A4*. This is represented by figure 3.16.

Case 2: Horizontal line of *A3A4* intersects with vertical line of *A1A2*. This is represented by figure 3.17.

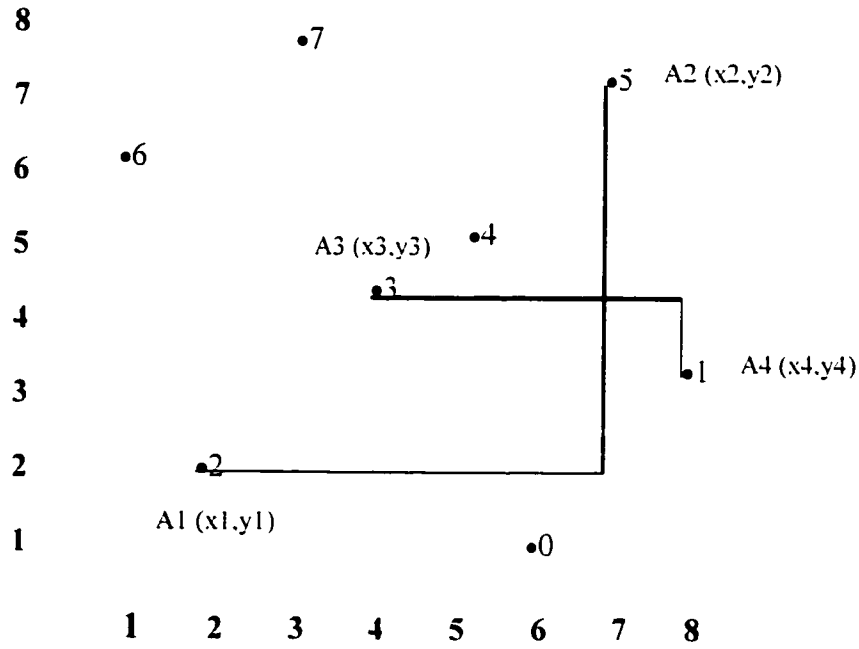


Fig. 3.17 horizontal line at A3 intersects with vertical line at A2

The following function checks whether an edge *A1A2* intersects with edge *A3A4*, with respect to the vertical-horizontal method.

```
Function Hor-Vert-Rook (A1,A2,A3,A4: integer; Px, Py :array[1..n] of integer)
                                    :boolean
Begin
    {obtain coordinates of edges A1A2 and A3A4}
    x1: = Px [A1]; y1: = Py [A1];
    x2: = Px [A2]; y2: = Py [A2];
    x3: = Px [A3]; y3: = Py [A3];
    x4: = Px [A4]; y4: = Py [A4];

    Hor-Vert-Rook: =false;

    if (MIN (x1,x2) < x4 AND x4 < MAX(x1,x2)) AND
       (MIN(y3,y4) < y1 AND  y1< MAX(y3,y4))
       then Hor-Vert-Rook: = true;  {case 1}

   if (MIN(x3,x4) < x2 AND x2 < MAX(x3,x4)) AND
     (MIN(y1,y2) <y3 AND y3 < MAX (y1,y2))
     then Hor-Vert-Rook: =true;  {case 2}
End.
```

3.6 Edge Intersections in Square Model

A square model for n processors is a (k x k) grid, with $k=\lceil \sqrt{n} \rceil$. Let the permutation of nodes be P = 4,7,2,5,10,1,8,3,9,6. The 4 x 4 square grid is shown in figure 3.17.

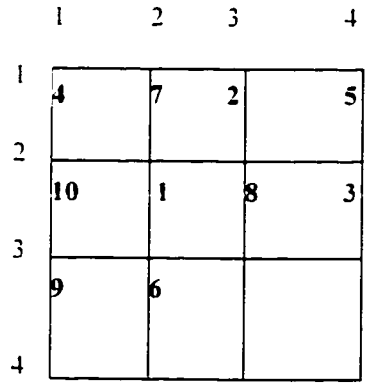


Fig. 3.18 4x4 grid

The table below shows the X and Y coordinates for the nodes represented by the index "I".

I	1	2	3	4	5	6	7	8	9	10
X	2	1	2	1	1	3	1	2	3	2
Y	2	3	4	1	4	2	2	3	1	1
P	4	7	2	5	10	1	8	3	9	6
IP	6	3	8	1	4	10	2	7	9	5

Fig. 3.19 Coordinates and permutations of nodes

The location for P [I] on the square grid is determined by the equations $row = I$ div k+1, and $column$ = [(I-1) mod k] +1

From the table, the inverse permutation IP of P is IP= 6, 3, 8, 1, 4, 10, 2, 7, 9, 5.

Now, the X,Y coordinates for the edges can be determined by the following equations:

$X[I] = (IP\ [I]$ div $k) + 1$;
$Y[I] = (\ IP[I] - 1$ mod $k\ ) + 1$

For example, for the edge (2,9),
The x-coordinate of 2 is (3 div 4) +1 = 1
The y-coordinate of 2 is (2 mod 4)+ 1 = 3
The x-coordinate of 9 is (9 div 4) +1= 3
The y-coordinate of 9 is (8 mod 4)+1 = 1

Hence, $(x1,y1) = (1,3)$ and $(x2,y2) = (3,1)$. These coordinates are used to determine whether or not two edges intersect, in the straight-line model for edge embedding.

The procedure *ConvertToCoordinates* converts the edges *A1A2* and *A3A4* to cartesion coordinates.

Procedure *ConvertToCoordinates (A1,A2,A3,A4:* ***integer****; var x1,y1,x2,y2,x3,y3,x4,y4:* ***integer****; P:* ***array****[1..n]* ***of integer****)*;
begin

InversePermutation (*P, IP*);

x1: = (*IP[A1]* div *k*)+1;
y1: = (*IP[A1]* -1 mod k)+1;

x2: = (*IP[A2]* div *k*)+1;
y2: = (*IP[A2]* -1 mod *k*)+1;

x3: = (*IP[A3]* div *k*)+1;
y3: = (*IP[A3]*-1 mod *k*)+1;

x4: = (*IP[A4]* div *k*)+1;
y4: = (*IP[A4]* -1 mod *k*)+1;

end.

The function *Straight-Line-Square* checks whether or not two edges *A1A2* and *A3A4* (where *A1,A2,A3,A4* correspond to nodes) in the straight-line method of edge embedding.

Function Straight-Line-Square (*A1, A2, A3, A4* : **integer** ; *P*: **array**[1..*n*] **of integer**): **boolean**
begin
ConvertToCoordinates *(A1,A2,A3,A4, x1,y1,x2,y2,x3,y3,x4,y4,P)*;
Straight-Line-Square : = Int-Edges (*x1, y1, x2, y2, x3, y3, x4, y4*);
end.

The horizontal-vertical method for edge embedding in the square model is not discussed in this thesis. Details of this model are left for future work.

4. HEURISTIC TECHNIQUES

In order to obtain experimental data about the pagenumber of an arbitrary graph, we use various heuristic techniques. Heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in achieving a particular goal. Heuristic search methods are used to determine solutions that take an infinite or very long time to compute. These methods might not always find the best solution, but are guaranteed to find a good solution in reasonable time. Two search methods, namely, Hill Climbing and Genetic Algorithms, are discussed.

4.1 Operators for permutations

The various heuristic methods make use of certain operators that are mandatory in creating robust search and optimization procedures that search for a global maximum or minimum. This section describes a few such operators for permutation oriented problems:

1. Selection : This operator selects permutations in the population for reproduction. The fitter the permutation, the higher its probability to be selected and reproduce.

PERMUTATION	FITNESS
A 1234	2
B 3214	1
C 4132	3

As permutation *C* has the maximum fitness, it has the highest probability to be selected.

2. Crossover : This operator randomly chooses a locus and exchanges the subsequences before and after the locus between two permutations to create two offspring. Some types of crossover are:

a. PMX (partially matched crossover): Given two parent permutations, *P1* and *P2*, two child permutations are created by dividing the two permutations into sections that define mappings between P1and P2 (marked by ||), exchanging one section, and then fixing the rest of each permutation by replacing corresponding elements. For example, let

P1 =(*jhd* || *efi* || *gcba*)
P2 =(*hge* || *bcj* || *iadf*)

Using the mid-section of *P1* and *P2* as a mapping (midsection of P*1* = *efi* and midsection of P*2* = *bcj*), replace *efi* in P*1* by *bcj* and *bcj* in P*2* by *efi*, we get

P1'= (*jhd* || *bcj* || *gcba*)
P*2'*=(*hge* || *efi* || *iadf*)

Replacing the extra cities *b.c.j* in *P1'* (in positions 9. 8 and 1) by *e*. *f* and *i* respectively.

P1''=(*ihd* || *bcj* || *gfea*),
and similarly.
P2''=(*hgb* || *efi* || *jadc*).

b. CX (cycle crossover): In cycle crossover. new strings are generated by finding cycles in the permutation formed by the two strings.

Consider the permutations:

P1=*jhdefigcba*
P2=*hgebcjiadf*

P1' is constructed by selecting the first element in permutation P*1*
P1'=(*j**********):

In the permutation. we see that *j* maps to *h*. and *h* maps to *g*. and *g* maps to *i*. and *i* maps to the first element *j*. completing a cycle. So all these are added to P1'. resulting in
P1'= (*jh*****ig****)

The empty slots (*) are filled in by the elements of P*2* at the corresponding positions. Thus.

P1'=(*jhebcigadf*). Similarly. P*2'* =(*hgdefjicba*).

c. Edge recombination operator : For each pair of individuals to be crossed. it (1) constructs a table of adjacencies in the parents. and (2). constructs one new permutation by combining information from the two parents:

- Select one parent at random and assign the first element in its permutation to be the first one in the child.
- Select the second element for the child. as follows: (a) if there is an adjacency common to both parents. then choose that element to be the next one in the child's permutation; (b) if there is an unused adjacency available from one parent. choose it; or if (a) and (b) fail. make a random selection.
- Select the remaining elements in order by repeating step 2.

For example, let the original individuals be 362145 and 521364.
The adjacency list is :

KEY	ADJACENT KEYS
1	2,2,3,4
2	1,1,5,6
3	1,6,6
4	1,5,6
5	2,4
6	2,3,3,4

Using the above algorithm, the new individual will be 364125.

3. Mutation: This operator rearranges randomly picked elements in a permutation. Some types of mutation are:

   a. swap
      This mutation chooses two elements and swaps them.
      For example, in the permutation 1234, if the elements 2 and 3 were selected, then the new permutation would be 1324.

   b. move block
      This mutation chooses a random block of elements and moves it to a point within the permutation.
      For example, for the permutation 1234, if the block is 34 and the position is 1, then the new permutation is 3412.

   c. swap blocks
      This mutation chooses two blocks and swaps them.
      For example for the permutation 12345, if the blocks were 12 and 45, then the new permutation would be 45312.

   d. insert
      Selects one element and inserts it at some other position in the permutation.
      For example, if the permutation is 34567 and the no 4 is selected with position 5, then the new permutation would be 35674.

   e. 2-opt
      Selects 2 points along a string, then reverses the segment between the points.
      For example, if the permutation is 12345, and the two points are 2 and 4, then the new permutation will be 14325.

   f. scramble
      Selects two points, and randomly reorders.

For example. Considering the permutation 12345 and the points 2and 4. the new permutation could be reordered as 13425.

We shall now describe a simple algorithm to generate a random permutation. It uses a function RANDOM( ) which returns a real number in the range [0.1]. selected at random. The selects one of first i elements at random and exchanges it with the i-th element. for $i = n, n\text{-}1,\ldots,1$.

```
Procedure Random-Perm( var Perm: array[1..n] of integer);
begin
 For i =1 to n do
    Perm [i] = i;
 For i = n downto 1 do
Begin
   j: = trunc( i* RANDOM( ))+1;
   Temp: = Perm[i];
   Perm [i]: = Perm [j];
   Perm [j]: = temp;
End
```

4.2 Hill Climbing

The Hill climbing problem solving technique is to evaluate the current solution and then move to a better solution closer to the goal. To determine whether another solution is better. some type of evaluation or reward function is needed. The solution is reached when the evaluation function is minimal. The evaluation function is usually referred to as the fitness function.

At first. we define an initial solution. say S_0. and calculate the fitness of S_0. The next step is to define one. few. or all neighbors of S_0. A neighbor of a permutation is obtained by applying any of the evolutionary computing operators on a candidate permutation. For example. if the initial solution S_0 = 1234. and the operator SWAP is applied at positions 2 and 4. then S_1 = 1432 will be the resulting neighbor. Hence. manipulating solutions with various evolutionary computing operators results in neighbors of the solution on which the operators were applied. We shall consider two instances of the Hill Climbing procedure.

In the first instance [WH], [P], a sample of neighbors of S_0 are generated (if the graph is relatively small. all neighbors may be generated). The fitness of all the neighbors is computed (each neighbor is evaluated). and the best neighbor (S') is determined. A move from S_0 to S' is then made. In the next iteration. a sample of neighbors of S' will be generated. This iterative procedure will continue till no better solution. compared to the current solution. can be found.

The algorithm is described below:

Choose an initial solution s in X
stop : =false
While *stop* =false **do**
Generate a sample V^* of solutions in $N(s)$ {$N(s)$ is the neighborhood of s}
Find the best s' in V^* {evaluate each neighbor and find s' such that
Fitness (s')=minimum among all neighbors}
if Fitness(s') > = Fitness(s)
then stop:= true
else s : = s'
end while

In the second instance [G1], a single neighbor (s') is generated in each iteration, and the fitness is computed. If s' is better than s_0, it becomes the next candidate permutation (a move from s_0 to s' is made), and in the next iteration, the neighbors of s' are generated. On the other hand, if the fitness is worse, then s' is ignored and another neighbor of s_0 is generated. A neighbor of a permutation is obtained by applying to the candidate permutation, one or more of the operators discussed in section 4.1. The process halts when no improvement is noticed.

S_0: = *initial solution;*
BestSolution: = S_0;
Fitness1: = F (S_0);
i: = 0;
repeat
C: = next neighbor (S_i) {one neighbor}
Fitness2: = $F(C)$;
if *fitness2* < *fitness1*
then
begin
BestSolution : = C;
S_{i+1} : = C; i: =i+1; *fitness1*: =*fitness2*;
end;
else
begin
S_{i+1} : = S_i ; i: =i+1;
end
until i=*max* OR S_i has no more neighbors

These procedures are used to look for the optimal solution in an optimization problem, but there is no guarantee that this is a global minimum.

Hill Climbing can be similarly used to search for the maximum (instead of minimum) in an optimization problem. It suffices to consider the additive or multiplicative inverse of the fitness function (that is, -fitness or 1/fitness).

4.3 Genetic Algorithms

The Genetic Algorithm refers to a model introduced and investigated by John Holland [H], and his students.

GA is a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures so as to preserve critical information.

An implementation of a GA begins with a population of (typically random) chromosomes. One then evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes, which represent a better solution to the target problem, are given more chances to "reproduce" than those chromosomes, which are poorer solutions.

In a broader usage of the term, a genetic algorithm is any population-based model that uses selection and recombination operators to generate new sample points in a search space. A simple GA is described below [SKM]:

```
Generate initial population P;
Evaluate (P);
While (not finished) do
   If ( SelectOperator = mutation)
      then
        Parent := SelectForMutation (P);
        Child := ReplicateAndMutate(parent);
      else
        **SelectOperator  =crossover**
        (father,mother):= SelectForCrossover(P);
        child := crossover (father,mother);
   endif
   Evaluate (child);
   Dead : = SelectDead (P);
  P[dead] := child;
Endwhile
Result := BestOf(P);
```

SelectForMutation randomly selects 2 different individuals from the population and returns the fittest. (This amounts to a selection pressure equal to a linear fitness ranking procedure, where the best individual has twice the probability to be selected, compared to the medium individual and the worst individual has zero probability.

SelectForCrossover selects at random 4 different individuals from the population, returning the 2 fittest for crossover.

SelectDead selects at random 3 different individuals and returns the least fit individual. This automatically creates an elite of two.
The various crossover and mutation operators on permutations have been described earlier.

5. EVOLUTIONARY COMPUTING FOR PAGENUMBER

5.1 Construction of pages

We consider the following four cases for embedding graphs:
Case 1: 1-D model
Case 2: 2-D grid model with the straight-line method of edge embedding
Case 3: 2-D rook model with the straight-line method of edge embedding
Case 4: 2-D rook model with the horizontal-vertical method of edge embedding

The solution space in cases 1 and 2 consist of a permutation of nodes N and an order of edges E, while cases 3 and 4 require a permutation of nodes N_X (in the X-direction), a permutation of nodes N_Y (in the Y- direction), and an order of edges E, in order to compute the pagenumber. The pair $N = (N_X, N_Y)$ of two permutation of nodes N_X and N_Y is considered as one permutation. Thus, a pair of permutations is simply called a permutation in order to provide uniform treatment for all four cases. The neighbors of $N=(N_X, N_Y)$ are $N' = (N'_X, N_Y)$ and $N'' = (N_X, N'_Y)$, where N_X' is a neighbor of N_X and N_Y' is neighbor of N_Y. The algorithm that searches for a page to place an edge remains the same for all four cases, however the condition that determines whether or not two edges cross changes.

The pagenumber of a graph may depend on the order in which the edges are selected to be placed on the pages; that is, they can be selected in lexicographic or random order. It is necessary to test both ways so as to ensure a minimal pagenumber.
For example, consider the (4x4) grid below:

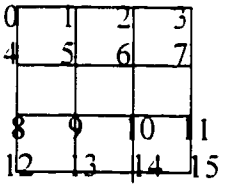


Fig. 5.1 4x4 grid

If the initial permutation of nodes, N= (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15), and the edges are selected in lexicographic order, then the pagenumber for the above permutation of nodes is 4.

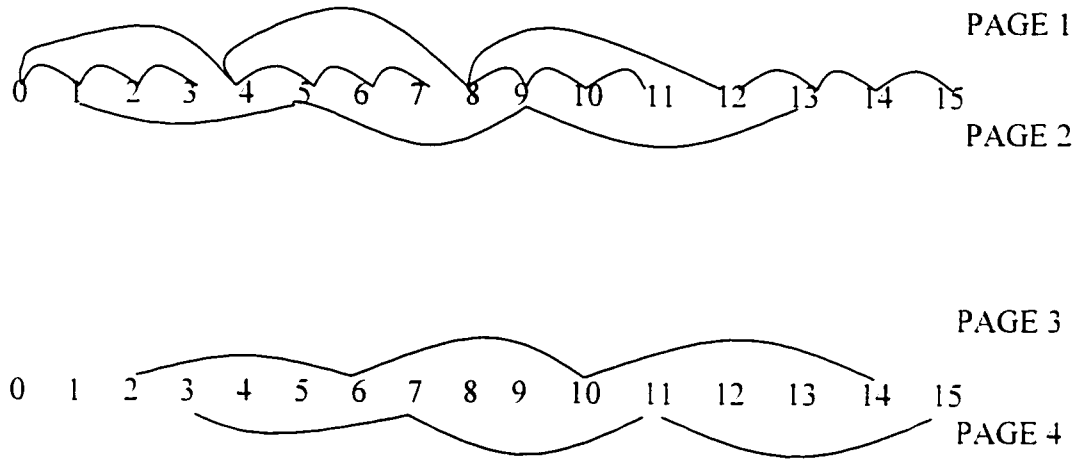


Fig. 5.2 4 page embedding

If the edges are selected in the following random order, then the pagenumber for the permutation N will be 5:
{(6,7),(4,5),(9,13),(7,11),(4,8),(2,3),(10,11),(11,15),(2,6),(5,9),(8,12),(12,13),(9,10),(5,6),(6,10),(8,9),(0,1),(1,5),(3,7),(13,14),(0,4),(14,15),(1,2),(10,14)}.

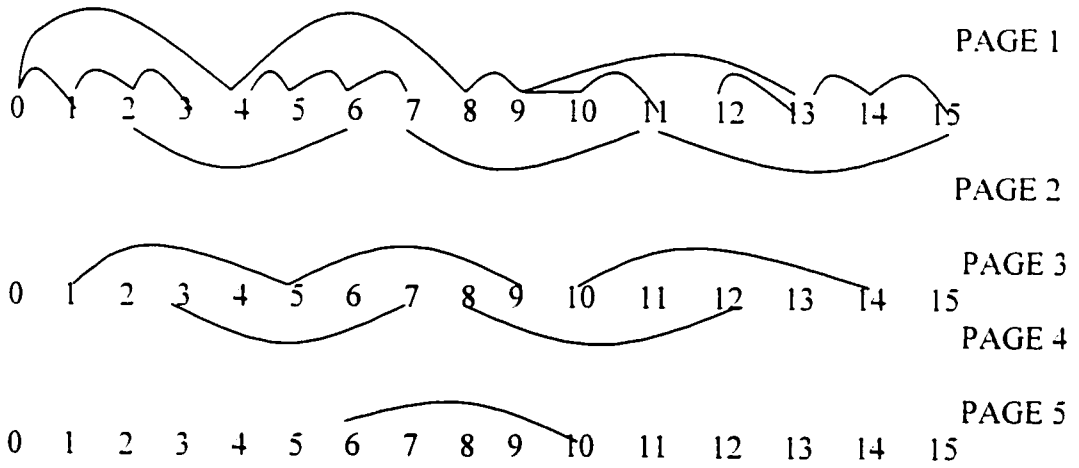


Fig. 5.3 5 page embedding

We shall now elaborate the function used to place edges on pages. The fitness function *Pagenumber (N)* is the minimal pagenumber for a given permutation of nodes, N, over all possible order of edges. However, it is not practical to test all orders of edges. Therefore *Pagenumber (N)* is approximated by testing a few edge orders, using a function *Pagenumber (N,E)* which finds pagenumber for node order N using edge order E. The pagenumber function is described briefly.

Function PN (N,E)

```
Begin
PageNo: = 1 (* initialize page number*)
For each edge e do
                    begin
                       i:=0;
                       repeat
                          i : =i+1;
                       until e can be placed on the i-th page;
                    if i:= PageNo+1 then PageNo: =PageNo+1
                    end;
End;
```

A (*no_of_edges* x 2) array, *E* stores the permutation of edges of the graph, and array N stores the permutation of nodes. $E[i,1]$ and $E[i,2]$ are end points of *i*-th edge, $1 \le i \le$ *no_of_edges*. The function *Pagenumber(N,E)* is specified next.

```
Function Pagenumber ( N:array[1..no_of_nodes] of integer, E:
                          array[1..no_of_edges,1..2] of integer): integer;
Begin
      Pagenumber: = 1;
      Initialize P1 with E[1,1] and E[1,2]: {first edge in E is embedded in page 1}
      for i: = 2 to  no_of_edges do
         begin
            a: = E[i,1]; b: = E[i,2]; {take next edge to be embedded}
            j: = 0;
            repeat
                  cross: =false;
                  j:=j+1; {go to next page}
                  while (NOT cross AND there are more edges in Pj) do
                     begin
                        (c,d): = next edge in Pj;
                        cross: = Intersect-1-Dim(a,b,c,d,N);
                     end
            until (NOT cross OR  j = Pagenumber)
            if cross
              then
                  begin
                     Pagenumber: = Pagenumber+1;
                     j: = Pagenumber;
                     add edge (a,b) to Pj; {create new page};
                  end
             else
                     add edge (a,b) to Pj
      end
End;
```

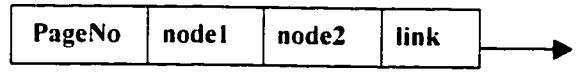
The line *cross: = Intersect-1-Dim(a, b, c, d, N)* in the above function will be replaced by the following lines, in order to find the pagenumber of a graph using the 2-D models:

- *cross*: = Straight-Line-Square(*a, b, c, d, N*), for the square model using the straight line method.
- *cross*: = Straight-Line-Rook (*a, b, c, d, Px,Py*), for the rook model using the straight line method.
- *cross*: = Hor-Vert-Rook (*a, b, c, d, Px, Py*), for the rook model using the horizontal-vertical method.

The above algorithm searches each page in order to determine if an edge (*a,b*) can be placed on that page. The edge will be placed if there occurs no crossing between (*a,b*) and any of the other edges already embedded in that page. However, if embedding the edge in every existing page results in a crossing, then a new page is created for it.

A linked list stores the pagenumber and the edges in each page. The linked list structure is defined as:

```
node = record
        PageNo : integer;
        node1 : integer;
        node2 : integer;
        link: ^node;
    end;
```



As an example, consider the graph in figure 5.4:

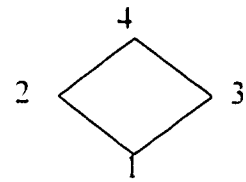


Fig. 5.4 graph with 4 vertices

Also, let us consider the candidate permutation to be *N* =1234. It is required to find the pagenumber of *N*. The *n*x2 (where *n* is the number of edges) array *E*, stores the edges of the graph in figure 5.4.

E

1	2
1	3
2	4
3	4

Fig. 5.5 array *E*

The first edge (1,2) is selected and placed on the first page. A corresponding entry is made in the linked list.

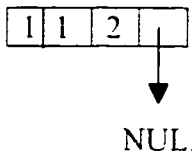


Fig 5.6 linked list with edge (1,2) on page 1

The first number in the linked list represents the page and the next two numbers represent the edge. In figure 5.6, edge (1,2) is placed on the first page.

The second edge (1,3) is selected, and it is determined whether it can be placed on the first page, without it crossing with the edge (1,2). As there is no crossing, (1,3) is placed in the first page.

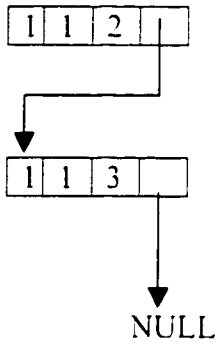


Fig 5.7 linked list with edge (1,3) on page 1

The next edge is (2,4). If we place it on the first page, there will be a crossing between (2,4) and (1,3). Therefore, (2,4) is placed on the next page.

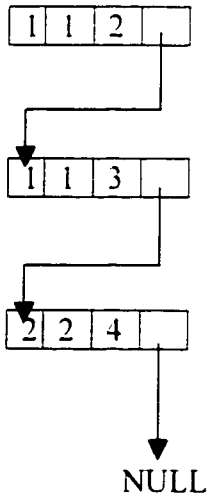


Fig 5.8 linked list with edge (2,4) on page 2

The last edge (3,4) can be placed on the first page, and so a node containing the edge and pagenumber is inserted after the edges (1,3) on page 1.

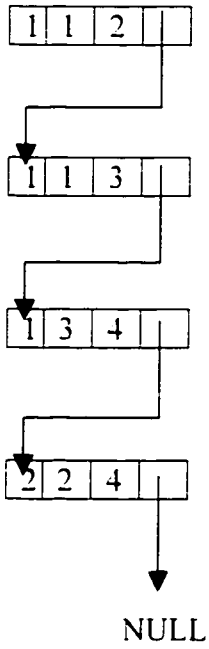


Fig 5.9 linked list with edge (3.4) on page 1

We will now elaborate on the *Pagenumber* function by referring to the linked list data structure. The function *PageCross* returns true if an edge $(a.b)$ crosses with any edge in page P_j. The procedure *NewPage* creates a new page to embed an edge $(a.b)$ which crosses in all existing pages. The procedure *AddEdge* adds an edge $(a.b)$ to a page P_j.

```
Function Pagenumber ( N:array[1..no_of_nodes] of integer; E:
                      array[1..no_of_Edges,1..2] of integer): integer;

var ptr, newnode, head : ^ node;
Begin
   ptr: = nil; newnode: =nil;
   Pagenumber: = 1;
   new ( head);
   head^.PageNo: = 1;
   head^.node1: = E[1,1];
   head^.node2: = E[1,2]; {first edge in E is embedded in page 1}
   head^.link: =nil;
   for i: = 2 to no_of_edges do
   begin
      a: =E[i,1]; b: =E[i,2]; {take next edge to be embedded}
      j: = 0;
      ptr: =head;
      repeat
            cross: =false;
            j: = j+1; {go to next page}
            while (ptr^.pageno< >j) do
                  ptr : = ptr^.link;
            while (ptr<>nil AND ptr^.pageno = j AND NOT cross ) do
              begin
                  c: = ptr^.node1;
                  d: = ptr^.node2;
                  cross: = Intersect-1-Dim (a, b,c,d,N);
                  ptr: = ptr^.link;
              end
      until (NOT cross OR j = Pagenumber)
      new(newnode);
      newnode^.node1: = a;
      newnode^.node2: = b;
      if cross then
                  begin
                     Pagenumber: = Pagenumber + 1;
                     j: = Pagenumber;
                     newnode^.PageNo: = j;
                     NewPage(head,newnode); {insert newnode at end of list}
                  end
               else
                  begin
                     newnode^.PageNo: = j;
                     AddEdge(j,head,newnode,pagenumber);
                  end
  end
```

```
    dispose(head); dispose(newnode);
End;
```

The procedures *NewPage* and *AddEdge* are defined below.

```
Procedure NewPage (var head: ^node; newnode: ^node);
var hptr: ^node;
Begin
    hptr: = head;
    while ( hptr^.link<>nil) do
            hptr: = hptr^.link;
    newnode^.link: =nil;
    hptr^.link: = newnode;
end;
```

```
Procedure AddEdge(j:integer; var head:^node; newnode: ^node; Pagenumber:integer);
var hptr, pres:^node;
Begin
    hptr: = head;
    if (j = Pagenumber)  {edge has to be inserted in the last page}
       then
          begin
             while (hptr^.link<>nil)
                   hptr: = hptr^.link;
             newnode^.link = nil;
             hptr^.link=node;
          end
       else
          begin
             while (hptr^.PageNo<> j+1) do
                begin
                   pres: = hptr;
                   hptr: = hptr^.link;
                end
             newnode^.link: = pres^.link;
             pres^.link: = newnode;
         end
End;
```

5.2 Direct Hill Climbing

Direct Hill climbing involves searching for a solution to the pagenumber problem, using the algorithms discussed in section 4.2. The solution space comprises of a permutation of nodes N, and an order of edges E. For the graph in figure 5.4, a solution, (N,E) can be (1 2 3 4, (1,2) (1,3) (2,4) (3,4)), where (1 2 3 4) is the permutation of nodes, N, and ((1,2) (1,3) (2,4) (3,4)) is the permutation of edges, E. Neighbors of the solution (N, E) are (N', E) and (N, E'), where N' is a neighbor of N and E' is a neighbor of E. The algorithm *HillClimbing* from section 4.2 is defined next, with respect to the Pagenumber function. This algorithm checks neighbors of N for a fixed order of edges E, until an improvement is noticed. If there is an improvement, the algorithm continues to evolve the new neighbor of N. The algorithm then checks all neighbors of E. The function NextNeighborN generates a neighbor of N, by using a mutation operator discussed in chapter 4. For example, one neighbor of (N,E) is (N',E) = (1 3 2 4, (1,2) (1,3) (2,4) (3,4)), where 2 and 3 in N have been swapped. Similarly, the function NextNeighborE generates a neighbor of E. For example, a neighbor of (N,E) is (N,E') = (1 2 3 4, (1,2) (2,4) (1,3) (3,4)), where (1,3) and (2,4) in E have been swapped.

```
Procedure HillClimbing;
Begin
   N0: = initial permutation of nodes;
   E0: = initial permutation of edges;
   PN1: = Pagenumber (N0,E0);
   BestSolution: = (N0,E0);
   i: = 0;
   N_Improvement: = false;
   repeat
          i: = i+1;
          (Ni', Ei): = NextNeighborN (BestSolution) {determine next neighbor of N}
          PN2: = pagenumber(Ni',Ei);
          if PN2 < PN1
                then
                   begin
                       BestSolution  : = (Ni',Ei);
                       PN1: = PN2;
                       N_Improvement: =true;
                   end;
  until no N_Improvement OR no more neighbors of N
  i: =0;
  E_Improvement : = false;
  repeat
          i: = i+1;
          (Ni, Ei'): = NextNeighborE (BestSolution) {determine a neighbor of E}
          PN2: = pagenumber(Ni,Ei');
          if PN2 < PN1
                then
                   begin
                       BestSolution  : = (Ni,Ei');
                       PN1: = PN2;
                       E_Improvement: =true;
                   end;

  until no E_Improvement OR no more neighbors of E
end;
```

5.3 Modified Hill Climbing

In Modified Hill Climbing, the solution also comprises of a permutation of nodes, N as well as an order of edges, E. The neighbor of a solution is obtained by evolving either the permutation of nodes or the permutation of edges. For example, for the graph in figure 5.4, an initial solution S could be (1234, {(1,2), (1,3), (2,4), (3,4)}). Examples of neighbors of S are {(2143, {(1,2), (1,3), (2,4), (3,4)}), (1234, {(1,3), (2,4), (1,2), (3,4)}, (1234, {(2,4), (1,3), (1,2), (3,4)}) }, etc., where either N or E has evolved.

The Hill Climbing algorithm MC computes the pagenumber for each N', with respect to a fixed order, E and random order, R of edges. It continues evolving N until there is no improvement or all neighbors of N have been checked. It then checks all neighbors of E. The function SwapNodes swaps elements i and j of N. Similarly, the function SwapEdges swaps elements p and q of E. The function SwapNodes and SwapEdges can be replaced with functions which make use of the other mutation operators discussed in chapter 4.

```
Algorithm MC
Begin
N: = initial permutation of nodes; E : = initial permutation of edges ;
PNUM= PN(N.E);
Repeat
        BETTER-FOUND: = FALSE; NO-MORE-NEIGHBORS-N: =FALSE;
        NO-MORE-NEIGHBORS-E: = FALSE;
        i: =1; j: =1;  { initialize elements to be swapped}
        Repeat
                X: = SwapNodes (N.i.j, NO-MORE-NEIGHBORS-N);
                PNXE: = PN(X.E);
                If PNXE <PNUM then
                                Begin
                                   BETTER-FOUND: = TRUE;
                                   PNUM: = PNXE;
                                   N: = X;
                                End;
                  R: = random order of edges;
                  PNXR: =PN(X.R);
                  If PNXR < PNUM then
                                    Begin
                                       BETTER-FOUND: = TRUE;
                                       PNUM: = PNXR;
                                       E: = R;
                                    End;
        Until BETTER-FOUND OR NO-MORE-NEIGHBORS-N
        BETTER: = FALSE;
        p: =1; q: =1;
        Repeat
                Y: = SwapEdges(E, p, q, NO-MORE-NEIGHBORS-E);
                PNNY: = PN(N.Y);
                If PNNY < PNUM then
                                Begin
                                   PNUM : = PNNY;
                                   E: = Y;
                                   BETTER: = TRUE;
                                   BETTER-FOUND: = TRUE;
                                End;
        Until BETTER OR NO-MORE-NEIGHBORS-E
Until NO BETTER-FOUND
End.
```

Other operators like insert and 2-op can also be applied to the nodes. In that case, we will call the functions InsertNodes (N) and 2-opNodes(N) to create neighbors of N.

We will use the swap operator only, in order to create neighbors of E. The functions SwapNodes (N,i,j) and SwapEdges(E,p,r) are defined next.

```
Function SwapNodes( N: array[1..m] of integer; var i, j : integer ; var no-neighbor-
                    n:boolean): array[1..m] of integer;
begin
    if ( j < m) then
                begin
                   j: =j+1
                   temp: = N[i];
                   N[i]: =N[j];
                   N[j]: = temp;
                   SwapNodes: = N;
                end
             else
                begin
                   if (i<>m)  then
                                 begin
                                   i: = i+1;
                                   j: =1;
                                   temp: = N[i];
                                   N[i]: =N[j];
                                   N[j]: = temp;
                                   SwapNodes: = N;
                                 end
                               else
                                   no-neighbor-n: =true;
                end
end;

Function SwapEdges(E: array[1..n,1..2] of integer; var p,q:integer; var no-neighbor-
                    e:boolean): array[1..n,1..2] of integer;
begin
   if ( q < n) then
                begin
                   q: =q+1;
                   temp1: = E[p,1]; temp2: =E[p,2];
                   E[p,1]: =E[q,1]; E[p,2]: =E[q,2];
                   E[q,1]: =temp1; E[q,2]: = temp2;
                   SwapEdges: = E;
                end
             else
                begin
                   if (p<>n) then
                                 begin
                                   p: = p+1;
                                   q: =1;
```

```
                    temp1: = E[p,1]; temp2: =E[p,2];
                    E[p,1]: =E[q,1]; E[p,2]: =E[q,2];
                    E[q,1]: = temp1; E[q,2]: = temp2;
                    SwapEdges: = E;
                end
            else
                no-neighbor-e: = true;
        end
end;
```

5.4 Pagenumber by Genetic Algorithms

In the case of genetic algorithms, a solution will also comprise of a permutation of nodes N and an order of edges E. However, in this case both the order of edges E and the permutation of nodes N are evolved simultaneously by an operator. For a solution $S(N,E)$ = (1 2 3 4, (1,2), (1,3), (2,4), (3,4)), a neighbor is created by evolving both N and E : $S(N',E')$ = (3 1 2 4, (1,3),(2,4),(1,2),(3,4)).

We begin by randomly creating a population P of solutions, where each solution $S_i = (N_i,E_i)$. In each generation, these solutions are evaluated, by calling the function Pagenumber (N_i,E_i), described in section 5.1. Since the problem is to minimize the function *pagenumber*, we will consider the additive inverse, - *pagenumber*. Assuming that the function pagenumber takes positive values on its domain, we will add a positive constant C (the worst pagenumber in the population), that is, a solution S_i is evaluated as $(-\ Pagenumber\ (N_i,E_i) + C)$. A new population is selected with respect to the probability distribution based on pagenumber values, and mutation and crossover operators are used to alter the solutions in the new population. After some number of generations, when no further improvement is observed, the best $S(N_i,E_i)$ in P represents an optimal solution.

For the selection process (selection of a new population with respect to the probability distributions based on the pagenumber values), the roulette wheel concept is used. Each individual (solution) in the population is given a slice of a circular roulette wheel that is inversely proportional in area to the individual's pagenumber (since the pagenumber is a minimization problem). The roulette wheel is constructed as follows:

- Find the total pagenumber of the population,

  $tot\text{-}pageno = \sum_{i=1}^{k} (-\ Pagenumber\ (N_i,E_i) + C')$, where k= number of individuals in P.

- Calculate the probability of selection, ps_i.
  $ps_i = (-\ Pagenumber\ (N_i,E_i) + C) / tot\text{-}pageno$, for all solutions S_i in P.
- Calculate the cumulative probability, q_i, for all solutions S_i in P.

Selection is based on spinning the roulette wheel *population-size* times, and generating a random number r between 0 and 1. If r is less than q_1, then the first solution (S_1) in P is selected, otherwise the i-th solution (S_i) is selected such that $q_{i-1} < r \leq q_i$. Obviously, some solutions would be selected more than once. The best solutions get more copies, the average stay even, and the worst die off.

Once the new populaton has been selected, crossover is performed to a number of individuals that are selected with a probability of crossover, *pc*. This probability gives us the expected number (*pc* * *population-size*) of individuals which undergo the crossover operation. A typical value for *pc* is 0.25 (25% of the individuals will undergo crossover). For each individual in the new population, we generate a random number r between 0 and 1. If r is less than *pc*, the given individual is selected for crossover. Next, the selected individuals are mated by picking father and mother pairs (in case the number of selected

individuals is odd, we can remove one extra individual, so that the remaining even individuals can be paired for crossover). Each pair of individuals yields two offspring. The parent individuals are replaced with the offspring, in the new population, *P*.

The next operator, mutation, is performed on an element-by-element basis. The total number of elements in a solution are *no-of-nodes* + *no-of-edges*. The probability of mutation, *pm*, gives us the expected number of mutated elements (*pm* * *m* * *population-size*). A typical value for *pm* is 0.01. For each individual (an individual comprises of *N* and *E*) in the current population, and for each element within the individual, we generate a random number, *r*, between 0 and 1. If *r* is less than *pm*, then that element is mutated with another randomly selected element in the same individual. For example, consider the individual S_i=(1 3 4 2, (1,2), (1,3), (2,4), (3,4)). The procedurs SwapN (*N,j,k*), and SwapE (E,*j,k*) are called to swap elements j and k in N or E. If the element *j*=3 is selected, then the element 4 in *N* will be swapped with a random generated ememt *k*, $1 \leq k \leq$ *no-of-nodes*. The function random (1,*no-of-nodes*) generates a random number between 1 and *no-of-nodes*. After swapping, we could get S_i'=(1 4 3 2, (1,2),(1,3),(2,4),(3,4)) as a neighbor of S_i. On the other hand, if the element *j* = 5 was selected, then (1,2) in *E* would be swapped with a randomly selected edge in position *k*, and we could get S_i =(1 3 4 2, (1,3), (1,2), (2,4), (3,4)). The function random(1, *no-of-edges*) generates a random number between 1 and *no-of-edges*.

Following selection, crossover, and mutation, the new population is ready for its next evaluation. The rest of the evolution is just cyclic repetition of selection, crossover, mutation, and evaluation. The genetic algorithm is defined next.

```
Generate initial population P:
m: = 0;
For i:= 1 to population-size do
      page-no_i: = – Pagenumber (N_i,E_j) + C
while (m< max-iterations) do
begin
    m: =m+1
    {construct roulette wheel}
    tot-pages: = 0;
    q_0 : = 0
     for i: = 1 to population-size do
                total-pages: = total-pages + page-no_i ;
    for i: = 1 to population-size do
        begin
            ps_i : = page-no_i / total-pages; { compute probability of selection}
            q_i : = q_i-1 + ps_i;
        end
    {spin roulette wheel to create new population P'}
    for i : = 1 to population-size do
        begin
            r : = random( ); { generates real number between 0 and 1}
            if r < q_1
               then P_i' : = P_1
               else
                  begin
                     k: =1;
                     repeat
                        k: = k+1;
                     until q_k-1 < r AND r < = q_k OR k > population-size
                     P_i': = P_k
                  end
        end
  t: =0;
  {select individuals for crossover}
   for i: = 1 to population-size do
   begin
       r: = random ( );
      if r < pc
        then  begin
                t: = t+1; C_t: = P_i'   {select P_i' for crossover }
                end
   end
  { select father and mother pairs in sequential order}
  k: =1
   while  k < t  do
                  begin
                    father: =C_k; mother: =C_k+1; k: =k+2;
                    ( child1,child2) : = crossover (father, mother);
                    P'( father ) : = child1; {replace parents by children}
                    P'( mother): =child 2;
                  end
```

```
{perform mutation}
 for i: = 1 to population-size do
      for j: = 1 to (no-of-nodes + no-of-edges) do
          begin
             r : = random( );
             if r < pm then
                          begin
                            if j < =no-of-nodes then
                                                  begin
                                                     k: = random(1,no-of-nodes)
                                                     swapN ( N,j,k )
                                                  end;
                                               else
                                                  begin
                                                    k: = random(1,no-of-edges);
                                                    swapE (E, j - no-of-nodes, k)
                                                  end
                          end
          end
  P : = P';
  For i: =1 to population-size do
       page-no_i: = (– Pagenumber (N_i,E_i) + C); { evaluate P}
end;
```

6. EXPERIMENTAL RESULTS

6.1 Some GA results

Mark Russell, undergraduate student at the University of Ottawa, conducted experiments for determining the pagenumber of several graphs, using genetic algorithms and the 1-D model for embedding edges. The initial population of solutions is generated either randomly or heuristically. The algorithm is terminated if the best fitness has not been surpassed for a given number of generations.

The graph types tested are shown in the left column. The next two columns give the number of nodes and edges of the graph being tested. The last column contains the pagenumber obtained for the graph with respect to the various crossover or mutation operators.

The tables 6.1 and 6.2 show the results obtained by using different embedding techniques. The embedding techniques embed the edges in different orders, often resulting in different pagenumbers. The orders used were:

- graphs are embedded by node - for each node in the graph, all edges incident to that node are embedded;
- graphs are embedded by path - from a randomly selected initial edge, edges are selected as much as possible to form a path;
- graphs are embedded randomly - the order in which edges are embedded is determined randomly;
- graphs are embedded by the three previous methods (node, path, and random) - the best book generated among these methods is selected.

The crossover operators, cyclic crossover (CX), edge recombination (ER), and partially matched crossover (PMX) are tested.

Table 6.1 shows the results obtained by using the insert operator for mutation, and the node embedding method. The population was randomly initialized.

Graph	**Nodes**	**Edges**	**Pagenumber**	**Best Pagenumber Known**	**Lower Bound**
Outerplanar	10	17	1	1	1
Square Grid	9	12	2	2	2
X-tree (depth = 4)	15	25	2	2	2
D-ary tree (D = 3)	13	12	1	1	1

Hypercube (d = 4)	16	32	3	3	d-1
Complete	10	45	8	5	n/2
FFT (d = 2)	12	16	2	3	3
Benes (d = 2)	16	24	2	3	3
Barrel Shifter (d = 2)	12	16	2	3	3

Table 6.1

Table 6.2 shows the results obtained by using the insert operator for mutation, and the path embedding method. The population was randomly initialized.

Graph	**Nodes**	**Edges**	**Pagenumber**	**Best Pagenumber Known**	**Lower Bound**
Outerplanar	10	17	1	1	1
Square Grid	16	24	2	2	2
X-tree (depth = 4)	15	25	2	2	2
D-ary tree (D = 3)	13	12	1	1	1
Hypercube (d = 4)	16	32	3	3	d-1
Complete	10	45	5	5	n/2
FFT (d = 2)	12	16	2	3	3
Benes (d = 2)	16	24	2	3	3
Barrel Shifter (d = 2)	12	16	2	3	3

Table 6.2

The same results are obtained by (i) using the insert operator for mutation, with the random embedding method, and (ii) using the insert operator for mutation, with the node, path, random embedding method. The population was randomly initialized. In these tests, the PMX crossover method provided the best overall results.

Table 6.3 is obtained by changing the initial generation of the population from random to heuristic. The heuristic method attempts to create a path between all adjacent nodes in the layout. The ER and PMX operators are tested.

Graph	**Nodes**	**Edges**	**Pagenumber**	**Best Pagenumber Known**	**Lower Bound**
Outerplanar	10	17	1	1	1
Square Grid	16	24	2	2	2
X-tree (depth = 4)	15	25	2	2	2
D-ary tree (D = 3)	13	12	1	1	1
Hypercube (d = 4)	16	32	3	3	d-1
Complete	10	45	5	5	n/2
DeBruijn (m = 2, k = 4)	16	29	3	3	3
FFT (d = 2)	12	16	2	3	3
Benes (d = 2)	16	24	2	3	3
Barrel Shifter (d = 2)	12	16	2	3	3

Table 6.3

Tables 6.4 and 6.5 show the results obtained with different mutation probabilities and mutation operators, and the CX operator. Table 6.4 shows the results for the insert operator with mutation probability of 20%. The same result is obtained with the scramble operator and mutation probability 40%.

Graph	Nodes	Edges	Pagenumber	Best Pagenumber Known	Lower Bound
D-ary tree (D = 3)	40	39	3	1	1
Hypercube (d = 5)	32	80	5	4	d-1

Table 6.4

Table 6.5 shows the results for the scramble operator with mutation probability of 20%. The same result is obtained with the insert operator and mutation probability 40%.

Graph	Nodes	Edges	Pagenumber	Best Pagenumber Known	Lower Bound
D-ary tree (D = 3)	40	39	3	1	1
Hypercube (d = 5)	32	80	6	4	d-1

Table 6.5

The above tables prove that the genetic algorithm is a reasonably good heuristic method for optimization problems. Optimal pagenumbers for outerplanar, square grids, and X-trees, Benes, FFT, and Barrel Shifter graphs were obtained in almost all tests.

6.2 1-D Results

The following tables comprise of the results obtained from testing the various algorithms and embedding techniques described in chapters 3, 4, and 5. Each table is characterized by the following attributes:

- Algorithm
  - Direct Hill Climbing (DC)
  - Modified Hill Climbing (MC)
  - Genetic Algorithm (GA)

- Model
  - 1-D
  - Square
  - Rook

- Embedding method
  - straight-line
  - horizontal-vertical

Table 6.6 shows the best known pagenumbers for graphs.

GRAPH	NODES *n*	EDGES *e*	BEST PAGENUMBER KNOWN	LOWER BOUND
Planar			4	4
Outerplanar			1	1
Square grid			2	2
X-tree			2	2
D-ary tree (D=2)	7	6	1	1
D-ary tree (D=3)	13	12	1	1
d-dimension hypercube (d=3)	8	12	2	d-1
d-dimension hypercube (d=4)	16	32	3	d-1
Complete graph	6	15	3	n/2

Table 6.6

The following tables show the results obtained for the 1-D model. The first column is the name of the graph being tested. The second column shows the number of nodes in each graph. The rest of the columns show the optimal pagenumbers (they are in bold if they correspond to the best pagenumbers) obtained by the various mutation or crossover operators.

Algorithm: DC
Model:1-D

GRAPH	NUMBER OF NODES	SWAP	INSERT	2-OPT
Planar	10	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**
Square grid	9	**2**	**2**	**2**
Square grid	16	4	4	4
X-tree	15	3	3	3
Binary tree	15	2	2	3
Hypercube (3-D)	8	**2**	**2**	**2**
Hypercube (4-D)	16	4	4	5
Hypercube (5-D)	32	7	7	9
Complete graph	6	**3**	**3**	**3**
DeBruijn graph	8	**2**	**2**	**2**

Table 6.7

Algorithm: MC
Model:1-D

GRAPH	NUMBER OF NODES	SWAP	INSERT	2-OPT
Planar	10	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**
Square grid	9	**2**	**2**	**2**
Square grid	16	4	4	4
X-tree	15	3	3	3
Binary tree	15	2	3	3
Hypercube (3-D)	8	**2**	**2**	**2**
Hypercube (4-D)	16	4	4	4
Hypercube (5-D)	32	9	9	8
Complete graph	6	**3**	**3**	**3**
DeBruijn graph	8	**2**	**2**	**2**

Table 6.8

Tables 6.7 and 6.8 show optimal results for outerplanar, square grid (n=9), 3-d hypercube, and the complete graph (n=6). All three operators performed equally well, in most cases, except in algorithm DC, where the swap and insert operators provided better results, and in the algorithm MC, where the 2-opt operator provided the best result for the 5-d hypercube.

Algorithm: GA
Model:1-D
Mutation Operator: Swap
Probability of mutation P_m: 0.2
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	**2**	**2**	**2**	**2**	**2**	**2**
Square grid	16	4	4	4	4	4	4
X-tree	15	4	4	4	4	4	4
Binary tree	15	3	3	3	3	2	3
Hypercube (3-D)	8	**2**	**2**	**2**	**2**	**2**	**2**
Hypercube (4-D)	16	5	5	5	5	5	5
Hypercube (5-D)	32	11	10	11	10	10	10
Complete graph	6	**3**	**3**	**3**	**3**	**3**	**3**
DeBruijn graph	8	3	**2**	**2**	**2**	3	3

Table 6.9

Algorithm: GA
Model:1-D
Mutation Operator: Insert
Probability of mutation P_m: 0.2
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	**2**	**2**	**2**	**2**	**2**	**2**
Square grid	16	4	4	4	4	4	4
X-tree	15	4	4	4	4	4	4
Binary tree	15	3	3	3	2	3	2
Hypercube (3-D)	8	**2**	**2**	**2**	**2**	**2**	**2**
Hypercube (4-D)	16	5	5	5	5	5	5
Hypercube (5-D)	32	10	9	10	10	9	10
Complete graph	6	**3**	**3**	**3**	**3**	**3**	**3**

DeBruijn graph	8	2	2	2	2	2	2

Table 6.10

The genetic algorithm did not perform as well as expected in the case of the 5-D hypercube. However, it showed the same minimal pagenumbers for the other graphs, as in the case of the DC and MC algorithms. There was no difference between the performance of the CX and PMX operators- both produced similar results.

6.3 2-D Results

The following tables show the results obtained from testing the grid and rook models and the two methods of edge embedding.

Algorithm: DC
Model: Square
Embedding method: straight-line

GRAPH	NUMBER OF NODES	SWAP	INSERT	2-OPT
Planar	10	2	2	2
Outerplanar	8	2	2	2
Square grid	9	1	1	1
Square grid	16	1	1	1
X-tree	15	3	2	3
Binary tree	15	2	2	2
Hypercube (3-D)	8	2	2	2
Hypercube (4-D)	16	3	3	3
Hypercube (5-D)	32	5	5	6
Complete graph	6	3	3	3
DeBruijn graph	8	2	2	2

Table 6.11

Algorithm: DC
Model: Rook
Embedding method: straight-line

GRAPH	NUMBER OF NODES	SWAP	INSERT	2-OPT
Planar	10	2	2	2
Outerplanar	8	1	1	1
Square grid	9	2	2	2

Square grid	16	4	4	4
X-tree	15	3	3	3
Binary tree	15	2	2	3
Hypercube (3-D)	8	2	2	2
Hypercube (4-D)	16	4	4	5
Hypercube (5-D)	32	8	8	9
Complete graph	6	3	3	3
DeBruijn graph	8	2	2	2

Table 6.12

Algorithm: DC
Model: Rook
Embedding method: horizontal-vertical

GRAPH	NUMBER OF NODES	SWAP	INSERT	2-OPT
Planar	10	2	2	2
Outerplanar	8	1	1	1
Square grid	9	2	2	2
Square grid	16	4	4	4
X-tree	15	2	2	3
Binary tree	15	2	2	3
Hypercube (3-D)	8	2	2	2
Hypercube (4-D)	16	3	4	4
Hypercube (5-D)	32	6	8	8
Complete graph	6	2	2	2
DeBruijn graph	8	2	2	2

Table 6.13

As in the case of 1-D, there are several cases shown in the tables where the swap and insert operators proved to be more effective compared to the 2-opt operator. The straight –line square method gave the best results for the square grid graph and the 5-*d* hypercube. The optimal embedding in the case of the square grid graph will always be one. This is due to the fact that the embedding model was the same as the graph-both were square grids. The two rook models gave a better result compared to the square model for the outerplanar graph, and the horizontal-vertical rook model gave the best result for the complete graph. The results of other graphs were similar for all three models.

Algorithm: MC
Model: Square
Embedding method: straight-line

GRAPH	NUMBER OF NODES	SWAP	INSERT	2-OPT
Planar	10	2	2	2
Outerplanar	8	2	2	2
Square grid	9	1	1	1
Square grid	16	1	1	1
X-tree	15	3	2	3
Binary tree	15	2	2	3
Hypercube (3-D)	8	2	2	2
Hypercube (4-D)	16	3	3	3
Hypercube (5-D)	32	5	5	6
Complete graph	6	3	3	3
DeBruijn graph	8	2	2	2

Table 6.14

Algorithm: MC
Model: Rook
Embedding method: straight-line

GRAPH	NUMBER OF NODES	SWAP	INSERT	2-OPT
Planar	10	2	2	2
Outerplanar	8	1	1	1
Square grid	9	2	2	2
Square grid	16	4	4	4
X-tree	15	3	3	3
Binary tree	15	3	3	3
Hypercube (3-D)	8	2	2	2
Hypercube (4-D)	8	4	4	5
Hypercube (5-D)	32	8	8	8
Complete graph	6	3	3	3
DeBruijn graph	8	2	2	2

Table 6.15

Algorithm: MC
Model: Rook
Embedding method: horizontal-vertical

GRAPH	NUMBER OF NODES	SWAP	INSERT	2-OPT
Planar	10	2	2	2
Outerplanar	8	1	1	1
Square grid	9	2	2	2
Square grid	16	4	4	4
X-tree	15	2	3	3
Binary tree	15	2	2	3
Hypercube (3-D)	8	2	2	2
Hypercube (4-D)	16	3	4	4
Hypercube (5-D)	32	6	8	7
Complete graph	6	2	2	2
DeBruijn graph	8	2	2	2

Table 6.16

Both the DC and MC algorithms gave the same result for the lowest pagenumber determined for all graphs, for example, both the algorithms yielded a pagenumber of 5 for the 5-dim hypercube (from the straight-line square model). Other results were similar to the DC algorithm.

In the case of the genetic algorithm, much of the result depended on the luck of the draw, after spinning the roulette wheel. Besides this, the other factors that affected the pagenumber were the crossover operators, mutation operators, and the two probabilities, *pc* and *pm*. All populations were randomly initialized, and the algorithm maintained a population size of 30 individuals. The following tables show the results from the genetic algorithm. The bold pagenumbers represent the minimum determined by the genetic algorithm, for a given method of edge embedding.

Algorithm: GA
Model: Square
Embedding method: straight-line
Mutation Operator: Swap
Probability of mutation P_m: 0.2
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	3	2	2	2	2	3
Outerplanar	8	2	2	2	2	2	2
Square grid	9	1	1	1	1	1	1
Square grid	16	1	1	1	1	1	1
X-tree	15	3	3	3	3	3	3
Binary tree	15	2	2	2	3	2	2
Hypercube (3-D)	8	2	2	2	2	2	2
Hypercube (4-D)	16	4	4	4	4	4	4
Hypercube (5-D)	32	6	6	6	6	6	6
Complete graph	6	3	3	3	3	3	3
DeBruijn graph	8	2	2	2	2	2	2

Table 6.17

Algorithm: GA
Model: Square
Embedding method: straight-line
Mutation Operator: Swap
Probability of mutation P_m: 0.4
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	3	3	3	3	3	3
Outerplanar	8	2	2	2	2	2	2
Square grid	9	1	1	1	1	1	1
Square grid	16	1	1	1	1	1	1
X-tree	15	3	3	4	4	4	4
Binary tree	15	2	2	2	2	2	2
Hypercube (3-D)	8	2	2	2	2	2	2
Hypercube (4-D)	16	4	4	4	4	4	4
Hypercube (5-D)	32	6	6	6	6	6	6

Complete graph	6	3	3	3	3	3	3
DeBruijn graph	8	2	2	2	2	2	2

Table 6.18

Algorithm: GA
Model: Square
Embedding method: straight-line
Mutation Operator: Insert
Probability of mutation P_m: 0.2
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	2	3	2	3	2	2
Outerplanar	8	2	2	2	2	2	2
Square grid	9	1	1	1	1	1	1
Square grid	16	1	1	1	1	1	1
X-tree	15	4	3	3	3	3	4
Binary tree	15	2	3	2	2	2	2
Hypercube (3-D)	8	2	2	2	2	2	2
Hypercube (4-D)	16	4	4	4	4	4	4
Hypercube (5-D)	32	6	6	6	6	6	6
Complete graph	6	3	3	3	3	3	3
DeBruijn graph	8	2	2	2	2	2	2

Table 6.19

Algorithm: GA
Model: Square
Embedding method: straight-line
Mutation Operator: Insert
Probability of mutation P_m: 0.4
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	2	2	2	2	2	2
Outerplanar	8	2	2	2	2	2	2
Square grid	9	1	1	1	1	1	1
Square grid	16	1	1	1	1	1	1
X-tree	15	4	3	2	3	3	4
Binary tree	15	2	2	3	3	2	3
Hypercube (3-D)	8	2	2	2	2	2	2
Hypercube (4-D)	16	4	4	4	4	4	4
Hypercube (5-D)	32	6	6	6	6	6	6
Complete graph	6	3	3	3	3	3	3
DeBruijn graph	8	2	2	2	2	2	2

Table 6.20

Table 6.20 shows that the combination of the insert mutation operator (pm=0.4) and the CX crossover operator (pc=0.3) gave the best result for the X-tree graph, with respect to the square straight-line method of edge embedding. For the rest of the graphs, these operators produced similar results.

Algorithm: GA
Model: Rook
Embedding method: straight-line
Mutation Operator: Swap
Probability of mutation P_m: 0.2
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	2	2	2	2	2	2
Outerplanar	8	1	1	1	1	1	1
Square grid	9	2	2	2	2	3	3
Square grid	16	4	4	4	4	4	4

X-tree	15	**3**	4	4	**3**	4	4
Binary tree	15	3	3	3	3	3	3
Hypercube (3-D)	8	**2**	**2**	**2**	**2**	**2**	**2**
Hypercube (4-D)	16	5	5	5	5	5	5
Hypercube (5-D)	32	**8**	10	10	10	9	10
Complete graph	6	**3**	**3**	**3**	**3**	**3**	**3**
DeBruijn graph	8	**2**	**2**	**2**	**2**	**2**	3

Table 6.21

Algorithm: GA
Model: Rook
Embedding method: straight-line
Mutation Operator: Swap
Probability of mutation P_m: 0.4
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	3	3	**2**	3	3	3
Square grid	16	**4**	**4**	**4**	**4**	**4**	**4**
X-tree	15	4	4	**3**	4	**3**	4
Binary tree	15	**2**	3	3	3	3	3
Hypercube (3-D)	8	3	**2**	**2**	**2**	**2**	3
Hypercube (4-D)	16	5	5	5	5	6	5
Hypercube (5-D)	32	10	9	10	9	10	10
Complete graph	6	**3**	**3**	**3**	**3**	**3**	**3**
DeBruijn graph	8	**2**	**2**	3	**2**	**2**	**2**

Table 6.22

Algorithm: GA
Model: Rook
Embedding method: straight-line
Mutation Operator: Insert
Probability of mutation P_m: 0.2
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	**2**	3	**2**	3	**2**	**2**
Square grid	16	**4**	**4**	**4**	**4**	**4**	**4**
X-tree	15	4	4	**3**	4	4	4
Binary tree	15	3	3	3	3	3	3
Hypercube (3-D)	8	**2**	**2**	**2**	3	3	**2**
Hypercube (4-D)	16	5	5	5	5	5	6
Hypercube (5-D)	32	10	10	10	10	10	10
Complete graph	6	**3**	**3**	**3**	**3**	**3**	**3**
DeBruijn graph	8	**2**	**2**	**2**	3	**2**	**2**

Table 6.23

Algorithm: GA
Model: Rook
Embedding method: straight-line
Mutation Operator: Insert
Probability of mutation P_m: 0.4
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	3	3	3	3	**2**	3
Square grid	16	**4**	**4**	**4**	**4**	**4**	**4**
X-tree	15	**3**	4	**3**	**3**	4	4
Binary tree	15	3	3	**2**	3	**2**	3
Hypercube (3-D)	8	**2**	3	**2**	**2**	3	3
Hypercube (4-D)	16	5	5	5	5	**4**	5

Hypercube (5-D)	32	9	10	10	10	10	10
Complete graph	6	**3**	**3**	**3**	**3**	**3**	**3**
DeBruijn graph	8	**2**	**2**	**2**	**2**	**2**	**2**

Table 6.24

Table 6.22 and 6.24 show that the best pagenumber obtained for the binary tree is 2. Table 6.24 shows a pagenumber of 4 for the 4-D hypercube. The operator and probability combinations in the rest of the tables do not produce a lower or equivalent pagenumber for these two graphs.

Algorithm: GA
Model: Rook
Embedding method: horizontal-vertical
Mutation Operator: Swap
Probability of mutation P_m: 0.2
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	**2**	**2**	**2**	**2**	**2**	**2**
Square grid	16	**3**	**3**	**3**	**3**	**3**	**3**
X-tree	15	**3**	**3**	**3**	**3**	**3**	**3**
Binary tree	15	**2**	**2**	3	**2**	**2**	**2**
Hypercube (3-D)	8	2	2	2	2	2	**1**
Hypercube (4-D)	16	4	4	4	4	4	4
Hypercube (5-D)	32	7	7	7	**6**	8	7
Complete graph	6	**2**	**2**	**2**	**2**	**2**	**2**
DeBruijn graph	8	2	2	2	2	2	**1**

Table 6.25

Algorithm: GA
Model: Rook
Embedding method: horizontal-vertical
Mutation Operator: Swap
Probability of mutation P_m: 0.4
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	**2**	**2**	**2**	**2**	**2**	**2**
Square grid	16	**3**	**3**	**3**	**3**	**3**	**3**
X-tree	15	**3**	**3**	**3**	**3**	**3**	**3**
Binary tree	15	**2**	**2**	**2**	**2**	**2**	**2**
Hypercube (3-D)	8	2	2	2	2	2	2
Hypercube (4-D)	16	4	4	4	4	4	4
Hypercube (5-D)	32	7	7	**6**	7	7	7
Complete graph	6	**2**	**2**	**2**	**2**	**2**	**2**
DeBruijn graph	8	2	2	2	2	2	2

Table 6.26

Algorithm: GA
Model: Rook
Embedding method: horizontal-vertical
Mutation Operator: Insert
Probability of mutation P_m: 0.2
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	**2**	**2**	**2**	**2**	**2**	**2**
Square grid	16	**3**	**3**	**3**	**3**	**3**	**3**
X-tree	15	**3**	**3**	**3**	**3**	**3**	**3**
Binary tree	15	**2**	**2**	3	**2**	**2**	**2**
Hypercube (3-D)	8	2	2	2	2	2	2
Hypercube (4-D)	16	4	**3**	4	**3**	**3**	**3**

Hypercube (5-D)	32	7	8	**6**	7	**6**	7
Complete graph	6	**2**	**2**	**2**	**2**	**2**	**2**
DeBruijn graph	8	2	2	2	2	2	2

Table 6.27

Algorithm: GA
Model: Rook
Embedding method: horizontal-vertical
Mutation Operator: Insert
Probability of mutation P_m: 0.4
Probability of Crossover Pc =0.1, 0.2, 0.3

GRAPH	NUMBER OF NODES	CX			PMX		
		Pc=0.1	Pc=0.2	Pc=0.3	Pc=0.1	Pc=0.2	Pc=0.3
Planar	10	**2**	**2**	**2**	**2**	**2**	**2**
Outerplanar	8	**1**	**1**	**1**	**1**	**1**	**1**
Square grid	9	**2**	**2**	**2**	**2**	**2**	**2**
Square grid	16	**3**	**3**	**3**	**3**	**3**	**3**
X-tree	15	**3**	**3**	**3**	**3**	**3**	**3**
Binary tree	15	**2**	**2**	**2**	**2**	**2**	**2**
Hypercube (3-D)	8	2	2	2	2	2	2
Hypercube (4-D)	16	4	4	**3**	4	**3**	4
Hypercube (5-D)	32	7	**6**	8	8	7	7
Complete graph	6	**2**	**2**	**2**	**2**	**2**	**2**
DeBruijn graph	8	2	2	2	2	2	2

Table 6.28

Table 6.25 shows that the best pagenumber obtained for the 3-D hypercube and deBruijn graph is 1. Both results are produced by the swap mutation operator (pm=0.2) and the PMX crossover operator (pc=0.3).

Table 6.29 is a comparison between the DC, MC, and GA algorithms for two-dimensional edge embedding methods. The rows show the various graphs with the same specifications as the ones in the tables shown earlier, and the columns represent the edge embedding method. For each graph and edge embedding method, the table shows the minimum pagenumber obtained and the algorithms that obtained them. The best pagenumbers obtained for each graph are in bold.

GRAPH	SQUARE *STRAIGHT-LINE*		ROOK *STRAIGHT-LINE*		ROOK *HORIZONTAL-VERTICAL*	
	Pagenumber	**Algorithm**	**Pagenumber**	**Algorithm**	**Pagenumber**	**Algorithm**
Planar	**2**	GA, DC, MC	**2**	GA, DC, MC	**2**	GA, DC, MC
Outerplanar	2	GA, DC, MC	**1**	GA, DC, MC	**1**	GA, DC, MC
Grid	**1**	GA, DC, MC	2	GA, DC, MC	2	GA, DC, MC
Grid	**1**	GA, DC, MC	4	GA, DC, MC	3	GA
X-tree	**2**	GA, DC, MC	3	GA, DC, MC	**2**	DC, MC
Binary tree	**2**	GA, DC, MC	**2**	GA, DC	**2**	GA, DC, MC
3-D hypercube	2	GA, DC, MC	2	GA, DC, MC	**1**	GA
4-D hypercube	**3**	DC, MC	4	GA, DC, MC	**3**	GA, DC, MC
5-D hypercube	**5**	DC, MC	8	GA, DC, MC	6	GA, DC, MC
Complete graph	3	GA, DC, MC	3	GA, DC, MC	**2**	GA, DC, MC
deBruijn	2	GA, DC, MC	2	GA, DC, MC	**1**	GA

Table 6.29

The square straight-line method showed the best results for the square grid and 4-D, 5-D hypercubes. The rook horizontal-vertical method performed best with the 3-D hypercube, complete graph, and the deBruijn graph.

Although the GA was expected to perform better than the other two algorithms (due to fact that the entire population was being evolved to produce fitter individuals, by using a combination of various crossover and mutation operators, and their probabilities), the only time it outperformed was with the rook horizontal-vertical technique used to embed the 16 node grid, 3-D hypercube, and the deBruijn graph.

The two hill climbing algorithms, DC and MC outperformed the GA thrice- twice with the square straight-line method (4-D, 5-D hypercubes), and once with the rook horizontal-vertical method (X-tree).

Further tests with genetic algorithms can be conducted by using other combinations of crossover and mutation operators, with varying probabilities.

The following tables give the timing of the various algorithms in seconds. Timing in seconds is obtained by dividing the processor time by the number of ticks per second.

Algorithm: DC
Model: 1-D
Mutation operator: swap

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.020
Outerplanar	8	0.020
Square grid	9	0.010
Square grid	16	0.120
X-tree	15	0.120
Binary tree	15	0.010
Hypercube (3-D)	8	0.010
Hypercube (4-D)	16	0.230
Hypercube (5-D)	32	8.743
Complete graph	6	0.010
DeBruijn graph	8	0.020

Table 6.30

Algorithm: MC
Model: 1-D
Mutation operator: swap

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.010
Outerplanar	8	0.010
Square grid	9	0.020
Square grid	16	0.140
X-tree	15	0.130
Binary tree	15	0.030
Hypercube (3-D)	8	0.010
Hypercube (4-D)	16	0.300
Hypercube (5-D)	32	9.884
Complete graph	6	0.020
DeBruijn graph	8	0.020

Table 6.31

Algorithm: GA
Model: 1-D
Mutation operator: swap
P_m: 0.2
P_c: 0.1

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.020
Outerplanar	8	0.020
Square grid	9	0.020
Square grid	16	0.030
X-tree	15	0.020
Binary tree	15	0.020
Hypercube (3-D)	8	0.020
Hypercube (4-D)	16	0.030
Hypercube (5-D)	32	0.110
Complete graph	6	0.010
DeBruijn graph	8	0.030

Table 6.32

Algorithm: DC
Model: Square
Mutation operator: swap
Embedding method: straight-line

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.030
Outerplanar	8	0.030
Square grid	9	0.040
Square grid	16	0.461
X-tree	15	0.371
Binary tree	15	0.060
Hypercube (3-D)	8	0.020
Hypercube (4-D)	16	0.881
Hypercube (5-D)	32	27.92
Complete graph	6	0.040
DeBruijn graph	8	0.041

Table 6.33

Algorithm: DC
Model: Rook
Mutation operator: swap
Embedding method: straight-line

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.020
Outerplanar	8	0.020
Square grid	9	0.010
Square grid	16	0.090
X-tree	15	0.100
Binary tree	15	0.020
Hypercube (3-D)	8	0.010
Hypercube (4-D)	16	0.220
Hypercube (5-D)	32	5.889
Complete graph	6	0.010
DeBruijn graph	8	0.010

Table 6.34

Algorithm: DC
Model: Rook
Mutation operator: swap
Embedding method: horizontal-vertical

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.030
Outerplanar	8	0.030
Square grid	9	0.040
Square grid	16	0.461
X-tree	15	0.371
Binary tree	15	0.060
Hypercube (3-D)	8	0.020
Hypercube (4-D)	16	0.881
Hypercube (5-D)	32	27.92
Complete graph	6	0.040
DeBruijn graph	8	0.041

Table 6.35

Algorithm: MC
Model: Square
Mutation operator: swap
Embedding method: straight-line

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.030
Outerplanar	8	0.040
Square grid	9	0.040
Square grid	16	0.581
X-tree	15	0.460
Binary tree	15	0.110
Hypercube (3-D)	8	0.030
Hypercube (4-D)	16	1.041
Hypercube (5-D)	32	30.92
Complete graph	6	0.050
DeBruijn graph	8	0.040

Table 6.36

Algorithm: MC
Model: Rook
Mutation operator: swap
Embedding method: straight-line

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.010
Outerplanar	8	0.010
Square grid	9	0.010
Square grid	16	0.130
X-tree	15	0.130
Binary tree	15	0.030
Hypercube (3-D)	8	0.010
Hypercube (4-D)	16	0.260
Hypercube (5-D)	32	6.719
Complete graph	6	0.020
DeBruijn graph	8	0.010

Table 6.37

Algorithm: MC
Model: Rook
Mutation operator: swap
Embedding method: horizontal-vertical

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.010
Outerplanar	8	0.010
Square grid	9	0.110
Square grid	16	0.160
X-tree	15	0.031
Binary tree	15	0.010
Hypercube (3-D)	8	0.250
Hypercube (4-D)	16	0.881
Hypercube (5-D)	32	6.269
Complete graph	6	0.020
DeBruijn graph	8	0.020

Table 6.38

Algorithm: GA
Model: Square
Embedding method: straight-line
Mutation operator: swap
P_m: 0.2
P_c: 0.1

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.030
Outerplanar	8	0.030
Square grid	9	0.030
Square grid	16	0.050
X-tree	15	0.050
Binary tree	15	0.020
Hypercube (3-D)	8	0.030
Hypercube (4-D)	16	0.040
Hypercube (5-D)	32	0.110
Complete graph	6	0.030
DeBruijn graph	8	0.020

Table 6.39

Algorithm: GA
Model: Rook
Embedding method: straight-line
Mutation operator: swap
P_m: 0.2
P_c: 0.1

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.020
Outerplanar	8	0.020
Square grid	9	0.020
Square grid	16	0.030
X-tree	15	0.030
Binary tree	15	0.020
Hypercube (3-D)	8	0.020
Hypercube (4-D)	16	0.030
Hypercube (5-D)	32	0.080

Complete graph	6	0.020
DeBruijn graph	8	0.020

Table 6.40

Algorithm: GA
Model: Rook
Embedding method:
horizontal-vertical
Mutation operator: swap
P_m: 0.2
P_c: 0.1

GRAPH	NUMBER OF NODES	TIME IN SECS
Planar	10	0.020
Outerplanar	8	0.020
Square grid	9	0.020
Square grid	16	0.030
X-tree	15	0.030
Binary tree	15	0.020
Hypercube (3-D)	8	0.020
Hypercube (4-D)	16	0.040
Hypercube (5-D)	32	0.110
Complete graph	6	0.020
DeBruijn graph	8	0.020

Table 6.41

CONCLUSION

We have implemented DC, MC, GA algorithms for the 1-D problem, and tested various graphs mentioned in the literature review. Optimal pagenumbers obtained for most graphs show that these algorithms are reasonably good heuristic methods for optimization problems.

We have introduced two 2-D layout models for embedding edges along with two edge embedding techniques. A number of experiments were conducted in order to ascertain that these models would yield reasonable pagenumbers.

8. FUTURE WORK

The pagenumber problem has been vastly studied with respect to the 1-D model. However, new models for embedding are yet to be studied and remain an interesting topic for further research. We have introduced the square and rook models for embedding edges. Using these two models, different edge- embedding techniques can be researched. Tests were conducted using the straight-line and horizontal-vertical method on the rook model, and the straight-line method on the square model. The horizontal-vertical method for the square model can be implemented for the various hill climbing and genetic algorithms.

Another fruitful area for further research is testing the rook, square, and other new models with different heuristic techniques such as tabu search and simulated annealing. Our results could be used as initial solutions in order to test these other techniques. More efficient crossover and mutation operators can be designed to further improve the pagenumber. The various methods and models for edge embedding can be tested with different types of graphs like star, pancake, higher dimension graphs, etc.

Further work on developing a better heuristic for initialization may improve results and place the layouts in a part of the search space more conducive to evolution.

9. REFERENCES

[AH] Nirwan Ansari and Edwin Hou. *Computational intelligence for optimization.* Kluwer Academic Publishers. 1996.

[BK] Bernhart F. and Kainen P.C.. The book thickness of a graph. *Journal of Combinatorial Theory – B*, 27. 1979. 320-331.

[BS] Jonathan F. Buss and Peter W. Shor. On the pagenumber of planar graphs. *Proc. of the Sixteenth Annual ACM Symposium On Theory of Computing*, 1984. 98-100.

[BT] Robert Battiti and Giampietro Tecchiolli. The reactive tabu search. *ORSA J. on Computing*, vol 6. 1994. 126-140

[CLR1] F.R.K. Chung. F.T. Leighton. and A.L. Rosenberg. Diogenes - A methodology for designing fault –tolerant processor arrays. *13th Intl. Conference on Fault Tolerant Computing*, 1983. 26-32.

[CLR2] Chung F.R.K . Leighton F.T. and Rosenberg A.L.. Embedding graphs in books- a layout problem with applications to VLSI design. *SIAM Journal on Algebraic Discrete Methods* . 8 . 1987. 33-58.

[dB] N.G. de Bruijn. A combinatorial problem. *Proc. Koninklijke Nederlandsche Akademie van Wetenschappen* (A) 49. Part 2. 1946. 758-764.

[EI] S. Even and A. Itai. Queues. stacks. and graphs. *Theory of Machines and Computations*. Z. Kohavi and A. Paz. eds.. Academic Press. New York. 1971. 71-86.

[G] Games R.A.. Optimal book embeddings of the FFT . Benes. and barrel shifter networks. *Algorithmica*. 1 . 1986. 233-250.

[G1] Glover F.. Tabu Search – Part-1. *ORSA J. on Computing* 1(3). 1989.

[G2] Glover F.. Tabu Search- Part –2 . *ORSA J. on Computing* 2(1). 1990.

[GH] Joseph L. Ganley. (Cadence Design Systems. Inc.. San Jose. California) and Lenwood S. Heath (Department of Computer Science. Virginia Polytechnic Institute and State University. Bblakburg, Virginia). The pagenumber of k-trees is o(k).

[GJMP] M.R. Garey. D.S. Johnson. G.L. Miller. and C.H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal of Algebraic Discrete Methods*. 1. 1980. 216-227.

[GKT] Galil Z., Kannan R. and Tarjan R.E., A separator theorem for graphs of bounded genus, *J. Algorithms* 5, 3, 1984, 391-407.

[H] Holland, J. Adaptation in natural and artificial systems, *University of Michigan Press*, 1975.

[H] Lenny Heath, Embedding planar graphs in seven pages, *25th Annual Symposium on Foundations of Computer Science*, October 1984, 74-83

[H2] L.S. Heath, Embedding outerplanar graphs in small books, *SIAM Journal of Algebraic Discreet Methods*, 8, 1987, 198-227.

[H3] Heath L.S., Algorithms for embedding graphs in books, Dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, N.C., 1985.

[HB] Handbook of Evolutionary computation, release 97/1, editors in chief: Thomas Back, David B. Fogel, Zbigniew Michalewics, published: Bristol: Philadelphia: NewYork: Institute of Physics Pub.: Oxford University Press, 1997

[HEU] HEURISCTIS (Intelligent Search Strategies for Computer Problem Solving), Judea Pearl, Addison-Wesley Publishing Company, 1984, page 35.

[HI] Lenwood S. Heath and Sorin Istrail, The pagenumber of genus g graphs is O(g), *Journal of the ACM*, 39(3), , July 1992, 479-501.

[I] S. Istrail, An algorithm for embedding planar graphs in six pages, *Manuscript*, 1986.

[JG] David S. Johnson, Lyle A. Mc Geoch, The travelling salesman problem: A case study in local optimization (a preliminary version of a chapter in the book *Local Search in Combinatorial Optimization*.E.H.L Aarts and J.K. Lenstra, John Wiley and Sons, London 1997, 215-310).

[K] C.D. Keys, Graphs critical for maximum bookthickness, *Pi Mu Epsilon Journal*, 6, 1975, 79-84.

[Ka] Kannan R., Unraveling k-page graphs, *Inf. Cont.*, 66, ½ (July/Aug.), 1-5.

[M] Melanie Mitchell, An introduction to genetic algorithms, *MIT Press*, 1996.

[MM] Seth M. Malitz, Genus g graphs have pagenumber O(sqrt(g)), *Journal of Algorithms*, 17(1):85-109, July 1994.

[MW] Moran and Y.Wolfstahl, One page book embedding under vertex-neighborhood constraints, *SIAM Journal of Discreet Mathematics*, 3, 1990, 376-390.

[MWW] Muder D.J., Weaver M.L. , and West D.B., Pagenumber of complete bipartite graphs, *Journal of Combinatorial Theory* 12, 1988, 469-489.

[MZ] Michalewicv, Z. *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd Edition, Springer-Verlag, New York, 1996.

[N] Alioune Ngom, Genetic algorithms for the jump number scheduling problem, *Order*, 15, 1, 1998, 59-73

[NP] Richard Nowakowski and Andrew Parker- Ordered sets, pagenumbers, and planarity, *Order*, volume 6, no.3 , 209-218, 1989.

[O] Obrenic B., Embedding deBrujin and shuffle-exchange graphs in 5 pages, *Proc. ACM SPAA* 1991, 137 – 146.

[Ol] L.T. Ollmann, On the book thickness of various graphs, *Proceedings, 4th S.E. Conference on Combinatorics, Graph Theory and Computing*, 1973, 459.

[R] Bruce Rosen, Function optimization based on advanced simulated annealing, Computer Science and Statistics, The University of Texas at San Antonio

[RS] R. Raghavan and S. Sahani, Single row routing, *IEEE Trans., Comput.*, C-32, 1983, 209-220.

[S] H.C. So, Some theoretical results on the routing of multilayer printed-wiring boards, *IEEE Intl. Symp. On Circuits and Systems*, 1974, 296-303.

[SGB] R. Swaminathan , D. Giraraj, and D.K. Bhatia , The Pagenumber of the class of bandwidth- k graph is k-1 , *Information Processing Letters* , 55 , 71-74 , 1995.

[SI] Stojmenovic I., Direct interconnection networks, in: *Parallel and Distributed Computing Handbook* (A.Y. Zomaya, ed.), McGraw-Hill, Inc., 1996, 537-567.

[SKM] Thomas Stidsen, V. Kragelund, Oana Mateescu, *Jobshop scheduling in a shipyard, 12th European Conference on Artificial Intelligence*, 1996.

[WH] D. de Werra and A. Hertz: Tabu Search Techniques, *OR Spectrum*(1989) 11: 131-141.

[Whit] Darell Whitley, A genetic algorithm tutorial, Computer Science Department, Colorado State University, 1993.

[Y] M. Yannakakis, Embedding planar graphs in four pages, *Journal of Computer and System Sciences* , 38 , 36-67 , 1989.

[Yo] Youngs, J.W.T., Minimal embeddings and the genus of a graph, *J. Math. Mech.*, 12, 2, 1963, 303-315.

10. APPENDIX

Code for the DC, MC, GA algorithms along with all the functions are found here. The following are functions called by the three algorithms.

File: inverse perm.c

```
/* finds the inverse permutation IP of P*/

void InversePermutation (int P[max_nodes],int IP[max_nodes], int
no_of_nodes)
/*Input: an array P which consists of a permutation of vertices.
 Output: the inverse permutation IP of P.*/
{
  int i;
  for (i = 0;i< no_of_nodes;i++)
        IP [P[i]] = i;
}
```

File 1-D.c

```
/*this function determines if there is an intersection between ab and
cd, using the 1-Dim model***/


bool Intersect_1_D (int aa,int bb,int cc, int  dd  )
{
   int min_ab,max_ab,min_cd,max_cd;
   bool Intersect;

   if( aa < bb)
            {
              min_ab  = aa ;  max_ab = bb;
            }
   else
            {
             min_ab = bb ; max_ab  = aa;
            }
   if (cc < dd)
            {
              min_cd  = cc ;  max_cd  = dd;
            }
   else
            {
             min_cd  = dd ; max_cd  = cc;
            }
   Intersect  = false;
   if (min_ab < min_cd &&  min_cd < max_ab && max_ab < max_cd)
          Intersect = true;
   if (min_cd < min_ab && min_ab < max_cd && max_cd < max_ab)
          Intersect = true;
 return Intersect;
}
```

File square.c

```
/******these functions determine whether or not lines in the square
model intersect*/

void ConvertToCoordinates(int A1,int A2,int A3,int A4,int
*x1,int*y1,int *x2,int* y2,int *x3,int *y3,int *x4, int*y4,int
N[max_nodes],int no_of_nodes)
{
 int IP[max_nodes];
 InversePermutation(N,IP,no_of_nodes);
 *x1=(int)(IP[A1]/k);
 *y1=( IP[A1]%k );

 *x2=(int)(IP[A2]/k);
 *y2=( IP[A2]%k);

 *x3=(int)(IP[A3]/k);
 *y3=( IP[A3] %k);

 *x4=(int)(IP[A4]/k);
 *y4=( IP[A4] %k);
}

bool SameSide(int x3,int y3,int x4,int y4,int x1,int y1,int x2,int y2)
{
  int nx,ny,vectorx,vectory;
  /* determine normal*/
  nx=y1-y2; ny=x2-x1;
  /*determine A2A4*/
  vectorx=x4-x2; vectory=y4-y2;
  if((nx*vectorx + ny*vectory)>0)
     return true;
  else return false;
}


bool Int_Edges(int x1,int y1,int x2,int y2,int x3,int y3,int x4,int y4)
{
 int a,b,c,d,e,f;
 float u;
 float v;
 a=x1-x2;
 b=x4-x3;
 c=x1-x3;
 d=y1-y2;
 e=y4-y3;
 f=y1-y3;

 if (a*e-b*d==0)
   {
      if (a*f-c*d==0) //overlapping lines
        {
          if (x1==x2)
              return (Intersect_1_D(y1,y2,y3,y4));
```

```
          else return(Intersect_1_D(x1,x2,x3,x4));
        }
      else return (false); //parallel lines
    }
  else
    {
      u=(float)((float)(c*e-b*f)/(float)(a*e-b*d));
      v=(float)((float)(a*f-c*d)/(float)(a*e-b*d));
      if( u<0 ||u>1 ||v<0||v>1) return false;
      if(u>0 &&u<1 &&v>0 &&v<1) return true; //intersecting lines
      if((u==0||u==1) && (v==0||v==1)) return false; //common end point
      if(u==0 &&v>0 &&v<1) return ( SameSide(x1,y1,x2,y2,x3,y3,x4,y4));
      if(u==1 &&v>0 &&v<1) return ( SameSide(x2,y2,x1,y1,x3,y3,x4,y4));
      if(v==0 &&u>0 &&u<1) return ( SameSide(x3,y3,x4,y4,x1,y1,x2,y2));
      if(v==1 &&u>0 &&u<1) return ( SameSide(x4,y4,x3,y3,x1,y1,x2,y2));
    }
    return(true);
  }



bool Straight_Line_Square(int A1,int A2,int A3,int A4,int
N[max_nodes],int no_of_nodes)
{
 int x1,y1,x2,y2,x3,y3,x4,y4;

ConvertToCoordinates(A1,A2,A3,A4,&x1,&y1,&x2,&y2,&x3,&y3,&x4,&y4,N,no_o
f_nodes);
 return(Int_Edges(x1,y1,x2,y2,x3,y3,x4,y4));
}
```

File: rookstraight.c

```
/*OBTAIN COORDINATES OF EDGES A1A2 AND A3A4*/

bool Straight_Line_Rook(int A1,int A2,int A3,int A4,int
Px[max_nodes],int Py[max_nodes],int no_of_nodes)
{

  int x1,y1,x2,y2,x3,y3,x4,y4;
  x1=Px[A1];y1=Py[A1];
  x2=Px[A2];y2=Py[A2];
  x3=Px[A3];y3=Py[A3];
  x4=Px[A4];y4=Py[A4];
  return(Int_Edges(x1,y1,x2,y2,x3,y3,x4,y4));
}
```

File: rookHV.c

```
/* these functions determine if edges intersect in rook model*/


int min(int x,int y)
{
 if (x<=y)return x;
 else return y;
}

int max(int x,int y)
{
 if(x>=y) return x;
 else return y;
}



bool Hor_Vert_Rook(int A1,int A2,int A3,int A4,int Px[max_nodes],int
Py[max_nodes],int no_of_nodes)
{
  //obtain coods of edges A1A2 and A3A4
   int x1,y1,x2,y2,x3,y3,x4,y4;
   bool intersection;

   x1=Px[A1]; y1=Py[A1];
   x2=Px[A2]; y2=Py[A2];
   x3=Px[A3]; y3=Py[A3];
   x4=Px[A4]; y4=Py[A4];

   intersection=false;

   if ( (min(x1,x2)<x4 && x4<max(x1,x2)) && (min(y3,y4)<y1 &&
y1<max(y3,y4)) )
      intersection=true;
   if ( (min(x3,x4)<x2 && x2<max(x3,x4)) && (min(y1,y2)<y3 &&
y3<max(y1,y2)) )
      intersection=true;
   return(intersection);
}
```

File: pagenumber.c

```
/***these functions determine the pagenumber of a graph*********/


bool Intersect_1_Dim(int a,int b,int c,int d, int N[max_nodes],int
no_of_nodes)
{
 int IP[max_nodes];
 InversePermutation(N,IP,no_of_nodes);
 return( Intersect_1_D(IP[a],IP[b],IP[c],IP[d]));
}


void NewPage (struct node** head, struct node * newnode)
{
     struct node * hptr;
     hptr = *head;
     while ( hptr->link!=NULL)
                 hptr = hptr->link;
     newnode->link =NULL;
     hptr->link = newnode;
}

void AddEdge(int j, struct node **head,struct node * newnode,int pages)
{
      struct node * hptr, *pres;
      hptr = *head;
      if (j == pages)  /*edge has to be inserted in the last page*/
          {
            while (hptr->link!=NULL)
                      hptr = hptr->link;
            newnode->link = NULL;
            hptr->link = newnode;
          }
      else
          {
            while (hptr->pageno != j+1)
                  {
                    pres = hptr;
                    hptr = hptr->link;
                  }
            newnode->link = pres->link;
            pres->link = newnode;
          }
}
```

```
int pagenumber(int N[max_nodes],int E[max_edges][2],int
Px[max_nodes],int Py[max_nodes],int no_of_nodes,int no_of_edges)

{
  int i,j,a,b,c,d,pages;
  struct node * ptr=NULL, *newnode=NULL, *head=NULL;
  bool cross;
  pages=1;
  head= (struct node *) malloc( sizeof(struct node));
  head->pageno=1;
  head->node1 = E[0][0];
  head->node2 = E[0][1]; /*first edge in E is embedded in page 1}*/
  head->link = NULL;
  for(i=1;i < no_of_edges;i++)
  {
   a =E[i][0];  b =E[i][1]; /*take next edge to be embedded*/
   j = 0;
   ptr =head;
   do
   {
       cross=false;
       j+=1;
       while(ptr->pageno!=j)
            ptr=ptr->link;
       while (ptr!=NULL && ptr->pageno == j && ! cross )
       {
         c = ptr->node1;
         d = ptr->node2;
         switch(method_choice)
         {
           case 1: cross = Intersect_1_Dim (a,b,c,d,N,no_of_nodes);
                   break;
           case 2: cross = Straight_Line_Square(a,b,c,d,N,no_of_nodes);
                   break;
           case 3: cross =
           Straight_Line_Rook(a,b,c,d,Px,Py,no_of_nodes); break;
           case 4: cross = Hor_Vert_Rook(a,b,c,d,Px,Py,no_of_nodes);
                   break;
         }
         ptr = ptr->link;
       }
  }while(cross && j!= pages);
  newnode=(struct node*) malloc(sizeof(struct node));
  newnode->node1 = a;
  newnode->node2 = b;
  if (cross)
          {
           pages = pages + 1;
           j = pages;
           newnode->pageno = j;
           NewPage(&head,newnode); //insert newnode at end of list//
          }
  else
          {
           newnode->pageno= j;
           AddEdge(j,&head,newnode,pages);
          }
```

```
  }

  free(head); free(newnode);
 /*printf("pagenumber %d",pages);*/
  return pages;
}
```

File: complete.c

```
/* this function computes the edges of a complete graph*/


int Complete(int E[max_edges][2],int no_of_nodes)
{
 int x=0,i,j;
 for(i=0;i<no_of_nodes;i++)
     for(j=0;j<no_of_nodes;j++)
       if(i!=j && i<j)
         {
          E[x][0]=i;
          E[x][1]=j;
          x++;
         }
       return x;
}
```

File: outerplanar.c

```
/* this function computes the edges of an outerplanar graph*/

int Outerplanar(int E[max_edges][2],int no_of_nodes)
{
 int x=0,i;
 for(i=1;i<no_of_nodes;i++)
    {
          E[x][0]=0;
          E[x][1]=i;
          x++;
          if(i<no_of_nodes-1)
             {
               E[x][0]=i;
               E[x][1]=i+1;
               x++;
             }
    }
       return x;
}
```

File: SquareGrid.c

```
/* this function computes the edges of a square grid*/

int SquareGrid(int E[max_edges][2],int no_of_nodes)
{
 int x=0,i,sqroot,row=1,col=1;
 sqroot=sqrt(no_of_nodes);
 for(i=0;i<no_of_nodes;i++)
    {
       if(col!=sqroot)
       {
          E[x][0]=i;
          E[x][1]=i+1;
          x++;
       }
       if(row!=sqroot)
       {
          E[x][0]=i;
          E[x][1]=i+sqroot;
          x++;
       }
       if(++col>sqroot)
        {
          col=1;
          row++;
        }
    }
       return x;
}
```

File: x_tree.c

```
/* this function computes the edges of a x_tree*/

int X_tree(int E[max_edges][2],int no_of_nodes)
{
 int x=0,i,row=1,pos=1;
 for(i=0;i<no_of_nodes;i++)
    {
       if(row< no_of_nodes/2)//not in bottom row
       {
          E[x][0]=i;
          E[x][1]=i*2+1;
          x++;
          E[x][0]=i;
          E[x][1]=i*2+2;
          x++;
       }
       if(row>1) //not in top row
       {
          if(pos!=row)
          {
```

```
            E[x][0]=i;
            E[x][1]=i+1;
            x++;
          }
       }
       if(++pos>row)
        {
          pos=1;
          row*=2;
        }
    }
       return x;
}
```

File: N_D_Tree

```
/* this function computes the edges of an n-vertex d-ary tree*/

int N_D_Tree(int E[max_edges][2],int no_of_nodes,int depth)
{
 int x=0,child,i;
 for(i=0;i<no_of_nodes;i++)
  {
     for(child=i*depth+1;child<=i*depth+depth;child++)
      {
        if(child>=no_of_nodes)break;
        E[x][0]=i;
        E[x][1]=child;
        x++;
      }
   }
 return x;
}
```

File: cube.c

```
/*these functions compute the edges of a d-dimendion cube*/

void init_nodes(int nodes[max_nodes][max_dim],int d, int no_of_nodes)
{
  int i,j,q,r,temp;

  for(i=0;i< no_of_nodes;i++)
    for(j=0;j<d;j++)
     nodes[i][j]=0;
  nodes[1][d-1]=1;

  for(i=2;i< no_of_nodes;i++)
  {  q=100; temp=i;
    for(j=d-1;j>=0&& q!=1;j--)
     {
      q=temp/2;
      r=fmod(temp,2);
      nodes[i][j]=r;
      if(q==1)
        nodes[i][j-1]=q;
      temp=q;
     }
  }
  for(i=0;i<no_of_nodes;i++)
  {
   for(j=0;j<d;j++)
       printf("%d ",nodes[i][j]);
   printf("\n");
  }

}



int reverse(int i,int d)
  {
     int x,j;
     int rev_array[max_dim];
     x=d-1;
     for(j=0;j<d;j++)
     {
       rev_array[j]=x;
       x--;
     }
     return rev_array[i];
   }
```

```
int bin_to_dec(int bits[max_dim],int bit,int d)
  {
    int total=0,i;
    for(i=0;i<d;i++)
    {
      if(bit==i)
        total+=pow(2,reverse(i,d));
      else
        total+=(bits[i]*pow(2,reverse(i,d)));
    }
  return total;
 }



int hypercube(int E[max_edges][2],int d,int no_of_nodes)
{
  int x=0,i,bit,j,dec;
  int nodes[max_nodes][max_dim];

  init_nodes( nodes,d,no_of_nodes);
  for(i=0;i<no_of_nodes;i++)
     for(bit=0;bit<d;bit++)
     {
      if(nodes[i][bit]==0)
       {
        E[x][0]=i;
        dec=bin_to_dec(nodes[i],bit,d);
        E[x][1]=dec;
        x++;
       }
     }
     for(i=0;i<x;i++)
     {
      for(j=0;j<2;j++)
       printf("%d ",E[i][j]);
      printf("\n");
     }

     return x;
}
```

File: RandomE.c

```
/*this function generates edges in random order*/

void RandomEdges(int E[max_edges][2],int R[max_edges][2],int
no_of_edges)
{
 int i,j=0;
 int temp[max_edges];
 for(i=0;i< no_of_edges;i++)
     temp[i]=random(2);//generate bit string
 for(i=0;i<no_of_edges;i++)
    {
      if(temp[i]==1)
        {
          R[j][0]=E[i][0];
          R[j][1]=E[i][1];
          j++;
        }
    }
 for(i=0;i<no_of_edges;i++)
    {
      if(temp[i]==0)
       {
        R[j][0]=E[i][0];
        R[j][1]=E[i][1];
          j++;
       }
    }
}
```

File: init_pop.c

```
/* this function generates an initial population for the GA*/

void InitialPopulation(int N[pop_size][max_nodes],int
E[pop_size][max_edges][2],int Px[pop_size][max_nodes],int
Py[pop_size][max_nodes],int no_of_nodes,int no_of_edges)
{
 int i,j,d;
 int temp1[max_edges],temp2[max_nodes];

 // create E population

 for(d=1;d<pop_size;d++)
 {
   j=0;
   for(i=0;i< no_of_edges;i++)
     temp1[i]=random(2);//generate bit string
   for(i=0;i<no_of_edges;i++)
    {
      if(temp1[i]==1)
        {
          E[d][j][0]=E[d-1][i][0];
          E[d][j][1]=E[d-1][i][1];
          j++;
```

```
        }
     }
    for(i=0;i<no_of_edges;i++)
     {
       if(temp1[i]==0)
        {
         E[d][j][0]=E[d-1][i][0];
         E[d][j][1]=E[d-1][i][1];
           j++;
        }
     }
 }

// create N,Px,Py population

for(d=1;d<pop_size;d++)
 { j=0;
   for(i=0;i< no_of_nodes;i++)
     temp2[i]=random(2);// generate bit string
   for(i=0;i<no_of_nodes;i++)
    {
      if(temp2[i]==1)
       {
        N[d][j]=N[d-1][i];
        Px[d][j]=Px[d-1][i];
        Py[d][j]=Py[d-1][i];
        j++;
       }
    }
   for(i=0;i<no_of_nodes;i++)
    {
      if(temp2[i]==0)
       {
        N[d][j]=N[d-1][i];
        Px[d][j]=Px[d-1][i];
        Py[d][j]=Py[d-1][i];
        j++;
       }
    }
 }
}
```

File: realrandom.c

```
/* this function returns a real number between 0 and 1 */

float realrandom()

{
   int x=random(pop_size+1);
   float y=(((float) x)/pop_size);
   return y;
}
```

File: copynodes.c

```
/* assigns one array of nodes to another*/

void AssignN1N2(int NextN[max_nodes],int N[max_nodes],int no_of_nodes)
 /* NextN[i]=N[0] */
{
 int i;
 for(i=0;i<no_of_nodes;i++)
   NextN[i]=N[i];
}
```

File: swapN.c

```
/*swaps nodes*/

void swapN (int NextN[max_nodes],int j,int ran)
{
 int temp1,temp2,temp;
 temp1=j;
 temp2=ran;
 temp=NextN[temp1];
 NextN[temp1]=NextN[temp2];
 NextN[temp2]=temp;
}
```

File:insertN.c

```
/*inserts a node*/

void insertN(int NextN[max_nodes],int j,int ran)
{
  int temp,i;
   if(ran>=j)
    {
      temp=NextN[ran];
      NextN[ran]=NextN[j];
      for(i=j;i<ran-1;i--)
        NextN[i]=NextN[i-1];
      NextN[i]=temp;
    }
   else
    {
      temp=NextN[ran];
      NextN[ran]=NextN[j];
      for(i=j;i>ran+1;i--)
        NextN[i]=NextN[i-1];
      NextN[i]=temp;
    }
}
```

File: copyedges.c

```
void AssignE1E2(int E1[max_edges][2],int E2[max_edges][2],int
no_of_edges)/*e2=e1*/

{
  int i;
  for( i=0;i<no_of_edges;i++)
  {
    E1[i][0]=E2[i][0];
    E1[i][1]=E2[i][1];
  }
}
```

File:swapE.c

```
/* this function swaps two edges*/

void swapE(int NextE[max_edges][2],int j,int k)
{
  int temp1,temp2,x,y;
  temp1=j;
  temp2=k;
  x=NextE[temp1][0];
  y=NextE[temp1][1];
  NextE[temp1][0]=NextE[temp2][0];
  NextE[temp1][1]=NextE[temp2][1];
  NextE[temp2][0]=x;
  NextE[temp2][1]=y;
}
```

File: copypopulation.c

```
/*assigns one population to the other*/

void AssignP1P2(int N[pop_size][max_nodes],int
NextN[pop_size][max_nodes],int E[pop_size][max_edges][2],int
NextE[pop_size][max_edges][2],int Px[pop_size][max_nodes], int
NextPx[pop_size][max_nodes], int Py[pop_size][max_nodes], int
NextPy[pop_size][max_nodes],int no_of_nodes,int no_of_edges)

{
 int i,j,k;

 //assign nodes

 for(i=0;i<pop_size;i++)
   for(j=0;j<no_of_nodes;j++)
    {
     NextN[i][j]=N[i][j];
     NextPx[i][j]=Px[i][j];
     NextPy[i][j]=Py[i][j];
    }

 //assign edges

 for(i=0;i<pop_size;i++)
   for(j=0;j<no_of_nodes;j++)
    for(k=0;k<2;k++)
     NextE[i][j][k]=E[i][j][k];
}
```

File: pmx.c

```
/*PMX CROSSOVER OPERATOR*/

void crossoverPMX(int father[max_nodes],int mother[max_nodes],int
child1[max_nodes],int child2[max_nodes],int no_of_nodes)
{
 int s1,s2,i,j,k,tempF[max_nodes],tempM[max_nodes],temp;
 AssignN1N2(child1,father,no_of_nodes);
 AssignN1N2(child2,mother,no_of_nodes);
 s1=random(no_of_nodes);
 s2=random(no_of_nodes);
 for(i=s1+1;i<=s2;i++)
 {
  temp=child1[i];
  child1[i]=child2[i];
  child2[i]=temp;
 }

 //child 1

 j=0;
 for(i=s1+1;i<=s2;i++)
```

```
{
  tempF[j]=child1[j];
  tempM[j]=child2[j];
  j++;
}

k=0;
while( k<j)
{
  for(i=0;i<=s1;i++)
   {
    if(tempF[k]==child1[i])
      child1[i]=tempM[k];
   }
  for(i>=s2+1;i<no_of_nodes;i++)
  {
    if(tempF[k]==child1[i])
      child1[i]=tempM[k];
  }
  k++;
}

//child 2

j=0;
k=0;
while( k<j)
{
  for(i=0;i<=s1;i++)
   {
    if(tempM[k]==child2[i])
      child2[i]=tempF[k];
   }
  for(i>=s2+1;i<no_of_nodes;i++)
  {
    if(tempM[k]==child2[i])
      child2[i]=tempF[k];
  }
  k++;
}
}
```

File: cx.c

```
/*these functions perform cyclic crossover*/

bool SearchInArray(int array[max_nodes], int search_pattern, int *pos,
int no_of_nodes)
{
  int i;
  for(i=0;i<no_of_nodes;i++)
      {
        if(search_pattern==array[i])
          {
            *pos=i;
             return(true);
          }
      }
  return(false);
}//end of function

void GetCycleCrossoverPositions(int parent1[max_nodes],int
parent2[max_nodes], int cycle_position[max_nodes],int no_of_nodes)
{
   int start_point,intermediate, pos=0;
   int i=0;
   start_point=parent1[0];
   intermediate=parent2[0];
   do
   {
     if(SearchInArray(parent1,intermediate,&pos,no_of_nodes))
       {
          intermediate=parent2[pos];
          cycle_position[i++]=pos;
       }
   }while(parent2[pos]!=start_point);
   cycle_position[i]=-1;
}//end of function

void crossoverCX(int father[max_nodes],int mother[max_nodes],int
child1[max_nodes],int child2[max_nodes],int no_of_nodes)
{
 int cycle_position[max_nodes];
 int i,temp;


 /*Create child1 and child2. Child1 has father as its dominant parent,
and
 child2 has mother. To begin with, let child1 and child2 take the first
element
 of their dominant parent*/

 child1[0]=father[0];
 child2[0]=mother[0];

 /*To determine the rest of the characteristics of the children, find
the
 positins that belong to a complete cycle. For details, on cyclic
```

```
 crossover, refer to chapter 4*/
 if(child1[0]!=child2[0])
  {

  GetCycleCrossoverPositions(father,mother,cycle_position,no_of_nodes);

    /* At each cycle position, child inherits the same node as that of
     its dominant parent.*/

    i=0;

    while( cycle_position[i]!=-1)
    {
     child1[cycle_position[i]]=father[cycle_position[i]];
     child2[cycle_position[i]]=mother[cycle_position[i]];
     i++;

    }

   /*rest of the elements are obtained from the other parent*/

   for(i=1;i<no_of_nodes;i++)
     if(SearchInArray(cycle_position,i,&temp,no_of_nodes)==false)
      {
       child1[i]=mother[i];
       child2[i]=father[i];
      }
 }
 else
  {
    for(i=1;i<no_of_nodes;i++)
     {
       child1[i]=mother[i];
       child2[i]=father[i];
    }
 }
}//end of function
```

File: swapnodes.c

```
/*****this function swaps nodes in order to form a neighbor*****/

 void SwapN(int N[max_nodes],int NextN[max_nodes],int*p1,int*p2,int
no_of_nodes)
 {
    int index,temp;
    if(method_choice==1||method_choice==2)
        {
         more_neighborsN=true;more_neighborsPxy=false;
        }
    else
        {
         more_neighborsPxy=true;more_neighborsN=false;
        }
    for(index=0;index<no_of_nodes;index++)
        NextN[index]=N[index];
    if(*p2<no_of_nodes-1)
       {
         (*p2)++;
         temp=NextN[*p1];
         NextN[*p1]=NextN[*p2];
         NextN[*p2]=temp;
       }
    else
       {
         if(*p1!=no_of_nodes-1)
           {
             (*p1)++;
             *p2=0;
             temp=NextN[*p1];
             NextN[*p1]=NextN[*p2];
             NextN[*p2]=temp;
           }
         else
            {
             if(method_choice==1||method_choice==2)
                more_neighborsN=false;
             else more_neighborsPxy=false;
            }
       }
 }
```

File: Opt2.c

```
/*****this function performs 2-OPT mutation in order to form a
neighbor*****/

 void OptN(int N[max_nodes],int NextN[max_nodes],int*p1,int*p2,int
no_of_nodes)
 {
    int index,temp,i,j;
    if(method_choice==1||method_choice==2)
        {
         more_neighborsN=true;more_neighborsPxy=false;
        }
    else
        {
         more_neighborsPxy=true;more_neighborsN=false;
        }
    for(index=0;index<no_of_nodes;index++)
        NextN[index]=N[index];
    if(*p1<no_of_nodes)
      {
          if(*p1>no_of_nodes-3)
           {
             if(*p1==no_of_nodes-2)
                *p2=0;
             else
                *p2=1;
           }
          else
            *p2=*p1+2;
          j=*p2;i=*p1;
          temp=NextN[i];
          NextN[i]=NextN[j];
          NextN[j]=temp;
          (*p1)++;
      }
    else
      {
         if(method_choice==1||method_choice==2)
            more_neighborsN=false;
         else more_neighborsPxy=false;
      }

}
```

File: insertnodes.c

```
/*****this function inserts nodes in order to form a neighbor*****/

 void InsertN(int N[max_nodes],int NextN[max_nodes],int*p1,int*p2,int
              no_of_nodes)
 {
    int index,temp,i;
    if(method_choice==1||method_choice==2)
        {
         more_neighborsN=true;more_neighborsPxy=false;
        }
    else
        {
         more_neighborsPxy=true;more_neighborsN=false;
        }
    for(index=0;index<no_of_nodes;index++)
        NextN[index]=N[index];
    if(*p2<no_of_nodes-1)
       {
         (*p2)++;
         if(*p2>=*p1)
           {
             temp=NextN[*p2];
             NextN[*p2]=NextN[*p1];
             for(i=*p1;i<(*p2)-1;i++)
               NextN[i]=NextN[i+1];
             NextN[i]=temp;
           }
         else

           {
             temp=NextN[*p2];
             NextN[*p2]=NextN[*p1];
             for(i=*p1;i>(*p2)+1;i--)
               NextN[i]=NextN[i-1];
             NextN[i]=temp;
           }
       }
    else
       {
         if(*p1!=no_of_nodes-1)
           {
               (*p1)++;
               *p2=0;
                if(*p2>=*p1)
                {
                  temp=NextN[*p2];
                  NextN[*p2]=NextN[*p1];
                  for(i=*p1;i<(*p2)-1;i++)
                    NextN[i]=NextN[i+1];
                  NextN[i]=temp;
                }
               else
               {
                  temp=NextN[*p2];
```

```
                    NextN[*p2]=NextN[*p1];
                    for(i=*p1;i>(*p2)-1;i--)
                       NextN[i]=NextN[i-1];
                    NextN[i]=temp;
                  }
              }
           else
              {
               if(method_choice==1||method_choice==2)
                  more_neighborsN=false;
               else more_neighborsPxy=false;
              }
        }
  }
```

File: swapedges.c

```
/*****this function swaps edges in order to form a neighbor*****/

 void SwapE(int E[max_edges][2],int NextE[max_edges][2],int
no_of_edges)
 {
    int index,temp1,temp2;
    more_neighborsE=true;
    for(index=0;index<no_of_edges;index++)
        {
          NextE[index][0]=E[index][0];
          NextE[index][1]=E[index][1];
        }

    if(pos2e<no_of_edges-1)
       {
         pos2e++;
         temp1=NextE[pos1e][0]; temp2=NextE[pos1e][1];
         NextE[pos1e][0]=NextE[pos2e][0];
NextE[pos1e][1]=NextE[pos2e][1];
         NextE[pos2e][0]=temp1; NextE[pos2e][1]=temp2;
       }
    else
       {
         if(pos1e!=no_of_edges-1)
           {
             pos1e++;
             pos2e=0;
             temp1=NextE[pos1e][0]; temp2=NextE[pos1e][1];
             NextE[pos1e][0]=NextE[pos2e][0];
NextE[pos1e][1]=NextE[pos2e][1];
             NextE[pos2e][0]=temp1; NextE[pos2e][1]=temp2;

           }
         else
            more_neighborsE=false;
       }
 }
```

File: DC.c

```
/************THIS PROGRAM USES THE DIRECT HILL CLIMBING METHOD TO
DETERMINE PAGENUMBERS*****/

/*define some preprocessor directives*/

#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<conio.h>
#include<math.h>

/* define some constant values and global variables*/

#define max_nodes 100
#define max_edges 500
#define max_dim 5

struct node
       {
         int pageno;
         int node1;
         int node2;
         struct node*link;
       };

bool more_neighborsN,more_neighborsE,more_neighborsPxy;
int pos1e,pos2e;

int method_choice,k;
int op_choice;

/* include all file required*/

#include<c:\thesis-1\Opt2.h>
#include<c:\thesis-1\insertnodes.h>
#include<c:\thesis-1\swapnodes.h>
#include<c:\thesis-1\swapedges.h>
#include<c:\thesis-1\inverse_perm.h>
#include<c:\thesis-1\1-D.h>
#include<c:\thesis-1\square.h>
#include<c:\thesis-1\rookstraight.h>
#include<c:\thesis-1\rookHV.h>
#include<c:\thesis-1\pagenumber.h>
#include<c:\thesis-1\complete.h>
#include<c:\thesis-1\Outerplanar.h>
#include<c:\thesis-1\SquareGrid.h>
#include<c:\thesis-1\x_tree.h>
#include<c:\thesis-1\N_D_Tree.h>
#include<c:\thesis-1\debruijn.h>
#include<c:\thesis-1\cube.h>
```

```
/* define main function*/

void main()
{
 int no_of_nodes,no_of_edges;
 int pos1n,pos2n,posPx1,posPx2,posPy1,posPy2,choice;
 int NextN[max_nodes],NextPx[max_nodes],NextPy[max_nodes];
 int NextE[max_edges][2];
 int N[max_nodes];
 int E[max_edges][2];
 int Px[max_nodes],Py[max_nodes];
 int PN1,PN2,i,graph,depth,m,b,dim;
 do
 {
 printf("enter graph 1->complete, 2->Outerplanar, 3->Square grid, 4-
>x_tree, 5-> N-vertex D-ary tree,6->debruijn,7->hypercube");
 scanf("%d",&graph);
 switch(graph)
 {
  case 1: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=Complete(E,no_of_nodes);
          break;
  case 2: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=Outerplanar(E,no_of_nodes);
          break;
  case 3: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=SquareGrid(E,no_of_nodes);
          break;
  case 4: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=X_tree(E,no_of_nodes);
          break;
  case 5: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          printf("\n enter depth");
          scanf("%d",&depth);
          no_of_edges=N_D_Tree(E,no_of_nodes,depth);
          break;
  case 6: printf("\n enter m and b");
          scanf("%d %d",&m,&b);
          no_of_nodes=pow(m,b);
          no_of_edges=debruijn(E,no_of_nodes,m,b);
          break;
  case 7: printf("\n enter dimensions");
          scanf("%d",&dim);
          no_of_nodes=pow(2,dim);
          no_of_edges=hypercube(E,dim,no_of_nodes);
          break;
 }
 printf("\n OPERATOR:1-> SWAP, 2->INSERT,3-> 2-OPT");
 scanf("%d",&op_choice);
 printf("\n METHOD: 1-> 1-D, 2 ->STRAIGHT_LINE_SQUARE, 3 -
>STRAIGHT_LINE_ROOK, 4->HOR_VERT_ROOK");
```

```
scanf("%d",&method_choice);
if (method_choice==2)
 {
   printf("\n ENTER K");
   scanf("%d",&k);
 }

// lexicographic initialization of nodes
for(i=0;i<no_of_nodes;i++)
  {
    N[i]=i;
    Px[i]=i;
    Py[i]=i;
  }

PN1=pagenumber(N,E,Px,Py,no_of_nodes,no_of_edges);

      pos1n=0;pos2n=0;posPx1=0;posPx2=0;posPy1=0;posPy2=0;
      do

         {
           switch(op_choice)
           {
               case 1: if (method_choice==1||method_choice==2)
                           SwapN(N,NextN,&pos1n,&pos2n,no_of_nodes);
                       else
                           {

                         SwapN(Px,NextPx,&posPx1,&posPx2,no_of_nodes);

                         SwapN(Py,NextPy,&posPy1,&posPy2,no_of_nodes);
                           }
                       break;

               case 2: if (method_choice==1||method_choice==2)
                           InsertN(N,NextN,&pos1n,&pos2n,no_of_nodes);
                       else
                           {

                       InsertN(Px,NextPx,&posPx1,&posPx2,no_of_nodes);

                       InsertN(Py,NextPy,&posPy1,&posPy2,no_of_nodes);
                           }
                       break;

               case 3: if (method_choice==1||method_choice==2)
                           OptN(N,NextN,&pos1n,&pos2n,no_of_nodes);
                       else
                           {

                        OptN(Px,NextPx,&posPx1,&posPx2,no_of_nodes);

                        OptN(Py,NextPy,&posPy1,&posPy2,no_of_nodes);
                           }
                      break;

         }
```

```
            PN2=
           pagenumber(NextN,E,NextPx,NextPy,no_of_nodes,no_of_edges);
            if(PN2 < PN1)
               {
                 if (method_choice==1||method_choice==2)
                  {
                    for(i=0;i<no_of_nodes;i++)
                       N[i]=NextN[i];
                  }
                 else
                    {
                     for(i=0;i<no_of_nodes;i++)
                     {
                       Px[i]=NextPx[i];Py[i]=NextPy[i];
                     }
                    }
                 PN1=PN2;

               }
          }while((more_neighborsN)||more_neighborsPxy);

       pos1e=0;pos2e=0;
       do
         {
           SwapE(E,NextE,no_of_edges);
           PN2= pagenumber(N,NextE,Px,Py,no_of_nodes,no_of_edges);
            if (PN2 < PN1)
               {
                 for(i=0;i<no_of_edges;i++)
                 {
                   E[i][0]=NextE[i][0];E[i][1]=NextE[i][1];
                 }
                 PN1=PN2;
               }
         }while(more_neighborsE);

    printf("pagenumber %d\n",PN1);
    if (method_choice==1||method_choice==2)
        for(i=0;i<no_of_nodes;i++)
            printf("%d\n ",N[i]);
    else
       {  for(i=0;i<no_of_nodes;i++)
             printf("%d ",Px[i]);
          printf("\n");
          for(i=0;i<no_of_nodes;i++)
             printf("%d ",Py[i]);

       }
    for(i=0;i<no_of_edges;i++)
       printf("(%d,%d)" , E[i][0],E[i][1]);
    printf("do you want to quit? 1=yes, 0=no");
    scanf("%d",&choice);
 }while(choice==0);
    getch();
}
```

File: MC.c

```
/*************THIS PROGRAM USES THE MODIFIED HILL CLIMBING METHOD TO
*****************DETERMINE GENUMBERS*********************************/

/* define preprosessor directives*/

#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<math.h>
#include<conio.h>

/*define some constants and global variables*/

#define max_nodes 100
#define max_edges 500
#define max_dim 5

struct node
       {
         int pageno;
         int node1;
         int node2;
         struct node*link;
       };

bool more_neighborsN,more_neighborsE,more_neighborsPxy;
int pos1e,pos2e;

int method_choice,k;
int op_choice;

/* include required files*/

#include<c:\thesis-1\Opt2.h>
#include<c:\thesis-1\insertnodes.h>
#include<c:\thesis-1\swapnodes.h>
#include<c:\thesis-1\swapedges.h>
#include<c:\thesis-1\RandomE.h>
#include<c:\thesis-1\inverse_perm.h>
#include<c:\thesis-1\1-D.h>
#include<c:\thesis-1\square.h>
#include<c:\thesis-1\rookstraight.h>
#include<c:\thesis-1\rookHV.h>
#include<c:\thesis-1\pagenumber.h>
#include<c:\thesis-1\complete.h>
#include<c:\thesis-1\Outerplanar.h>
#include<c:\thesis-1\SquareGrid.h>
#include<c:\thesis-1\x_tree.h>
#include<c:\thesis-1\N_D_Tree.h>
#include<c:\thesis-1\debruijn.h>
#include<c:\thesis-1\cube.h>
```

```
/*main function*/

void main()
{
 int no_of_nodes,no_of_edges;
 int pos1n,pos2n,posPx1,posPx2,posPy1,posPy2,choice;
 int NextN[max_nodes],NextPx[max_nodes],NextPy[max_nodes];
 int NextE[max_edges][2];
 int N[max_nodes];
 int E[max_edges][2];
 int R[max_edges][2];
 int Px[max_nodes],Py[max_nodes];
 int PN1,PNXE,PNXR,PNNE,i,graph,depth,m,b,dim;
 do
 {
  printf("enter graph 1->complete, 2->Outerplanar, 3->Square grid, 4-
>x_tree, 5-> N-vertex D-ary tree,6->debruijn,7->hypercube");
  scanf("%d",&graph);
  switch(graph)
 {
  case 1: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=Complete(E,no_of_nodes);
          break;
  case 2: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=Outerplanar(E,no_of_nodes);
          break;
  case 3: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=SquareGrid(E,no_of_nodes);
          break;
  case 4: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=X_tree(E,no_of_nodes);
          break;
  case 5: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          printf("\n enter depth");
          scanf("%d",&depth);
          no_of_edges=N_D_Tree(E,no_of_nodes,depth);
          break;
  case 6: printf("\n enter m and b");
          scanf("%d %d",&m,&b);
          no_of_nodes=pow(m,b);
          no_of_edges=debruijn(E,no_of_nodes,m,b);
          break;
  case 7: printf("\n enter dimensions");
          scanf("%d",&dim);
          no_of_nodes=pow(2,dim);
          no_of_edges=hypercube(E,dim,no_of_nodes);
          break;
 }
printf("\n OPERATOR:1-> SWAP, 2->INSERT,3->2-OPT");
scanf("%d",&op_choice);
```

```
printf("\n METHOD: 1-> 1-D, 2 ->STRAIGHT_LINE_SQUARE, 3 -
>STRAIGHT_LINE_ROOK, 4->HOR_VERT_ROOK");
scanf("%d",&method_choice);
if (method_choice==2)
 {
   printf("\n ENTER K");
   scanf("%d",&k);
 }

// lexicographic initialization of nodes
for(i=0;i<no_of_nodes;i++)
  {
    N[i]=i;
    Px[i]=i;
    Py[i]=i;
  }

PN1=pagenumber(N,E,Px,Py,no_of_nodes,no_of_edges);

      pos1n=0;pos2n=0;posPx1=0;posPx2=0;posPy1=0;posPy2=0;
      do

         {
           switch(op_choice)
           {
               case 1: if (method_choice==1||method_choice==2)
                           SwapN(N,NextN,&pos1n,&pos2n,no_of_nodes);
                       else
                           {

                            SwapN(Px,NextPx,&posPx1,&posPx2,no_of_nodes);

                            SwapN(Py,NextPy,&posPy1,&posPy2,no_of_nodes);
                           }
                       break;

               case 2: if (method_choice==1||method_choice==2)
                           InsertN(N,NextN,&pos1n,&pos2n,no_of_nodes);
                       else
                           {

                       InsertN(Px,NextPx,&posPx1,&posPx2,no_of_nodes);

                       InsertN(Py,NextPy,&posPy1,&posPy2,no_of_nodes);
                           }
                       break;

               case 3:  if (method_choice==1||method_choice==2)
                           OptN(N,NextN,&pos1n,&pos2n,no_of_nodes);
                       else
                           {

                          OptN(Px,NextPx,&posPx1,&posPx2,no_of_nodes);

                          OptN(Py,NextPy,&posPy1,&posPy2,no_of_nodes);
                           }
                       break;
```

```
        }
        PNXE=
        pagenumber(NextN,E,NextPx,NextPy,no_of_nodes,no_of_edges);
        if(PNXE < PN1)
           {
            if (method_choice==1||method_choice==2)
                {
                  for(i=0;i<no_of_nodes;i++)
                     N[i]=NextN[i];
                }
            else
               {
                   for(i=0;i<no_of_nodes;i++)
                   {
                     Px[i]=NextPx[i];Py[i]=NextPy[i];
                   }
               }
            PN1=PNXE;

           }
        RandomEdges(E,R,no_of_edges);
        PNXR=
        pagenumber(NextN,R,NextPx,NextPy,no_of_nodes,no_of_edges);
        if(PNXR < PN1)
            {
             for(i=0;i<no_of_edges;i++)
             {
               E[i][0]=R[i][0];E[i][1]=R[i][1];
             }
             PN1=PNXR;
            }

      }while((more_neighborsN)||more_neighborsPxy);

   pos1e=0;pos2e=0;
   do
     {
       SwapE(E,NextE,no_of_edges);
       PNNE= pagenumber(N,NextE,Px,Py,no_of_nodes,no_of_edges);
        if (PNNE < PN1)
           {
             for(i=0;i<no_of_edges;i++)
             {
               E[i][0]=NextE[i][0];E[i][1]=NextE[i][1];
             }
             PN1=PNNE;
           }
     }while(more_neighborsE);

printf("pagenumber %d\n",PN1);
if (method_choice==1||method_choice==2)
    for(i=0;i<no_of_nodes;i++)
        printf("%d\n ",N[i]);
else
   {  for(i=0;i<no_of_nodes;i++)
         printf("%d ",Px[i]);
```

```
            printf("\n");
            for(i=0;i<no_of_nodes;i++)
               printf("%d ",Py[i]);

         }
      for(i=0;i<no_of_edges;i++)
         printf("(%d,%d)" , E[i][0],E[i][1]);
      printf("do you want to quit? 1=yes, 0=no");
      scanf("%d",&choice);
}while(choice==0);
      getch();
}
```

File: GA.c

```
/****THIS PROGRAM USES THE GNETIC ALGORITHM METHOD TO DETERMINE*******
*******************PAGENUMBERS *************/

/*define preprocessor directives*/

#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<math.h>
#include<conio.h>

/* define constants and global variables*/

#define max_nodes 100
#define max_edges 500
#define max_dim 5
#define pop_size 30
#define max_iterations 2

struct node
       {
         int pageno;
         int node1;
         int node2;
         struct node*link;
       };


int method_choice;
int k; //can also be declared as double for ceil function
int op_choice;

/*include required files*/

#include<c:\thesis-1\inverse_perm.h>
#include<c:\thesis-1\init_pop.h>
#include<c:\thesis-1\1-D.h>
#include<c:\thesis-1\square.h>
#include<c:\thesis-1\rookstraight.h>
#include<c:\thesis-1\rookHV.h>
#include<c:\thesis-1\pagenumber.h>
#include<c:\thesis-1\realrandom.h>
#include<c:\thesis-1\copynodes.h>
#include<c:\thesis-1\swapN.h>
#include<c:\thesis-1\insertN.h>
#include<c:\thesis-1\copyedges.h>
#include<c:\thesis-1\swapE.h>
#include<c:\thesis-1\copypopulation.h>
#include<c:\thesis-1\pmx.h>
#include<c:\thesis-1\cx.h>
#include<c:\thesis-1\complete.h>
#include<c:\thesis-1\Outerplanar.h>
#include<c:\thesis-1\SquareGrid.h>
#include<c:\thesis-1\x_tree.h>
```

```
#include<c:\thesis-1\N_D_Tree.h>
#include<c:\thesis-1\debruijn.h>
#include<c:\thesis-1\cube.h>


/*main function*/

void main()
{
 int choice, mut_choice,cross_choice;
 int NextN [pop_size][max_nodes],NextPx[pop_size][max_nodes],NextPy
[pop_size][max_nodes];
 int NextE[pop_size][max_edges][2];
 int
N[pop_size][max_nodes],c[pop_size][max_nodes],cx[pop_size][max_nodes],c
y[pop_size][max_nodes],index[pop_size];
 int E[pop_size][max_edges][2];
 int Px[pop_size][max_nodes],Py[pop_size][max_nodes];
 int PN[pop_size], PageNo[pop_size];
 float ps[pop_size],q[pop_size];
 int
father[max_nodes],mother[max_nodes],child1[max_nodes],child2[max_nodes]
;
 int
fatherx[max_nodes],motherx[max_nodes],child1x[max_nodes],child2x[max_no
des];
 int
fathery[max_nodes],mothery[max_nodes],child1y[max_nodes],child2y[max_no
des];
 int i,j,m,tot_pages,l,n,t,ran,min_pages;
 float r,pm,pc;
 int graph,m_debruijn,b,depth,dim;
 int no_of_nodes,no_of_edges;
 do
 {
 printf("enter graph 1->complete, 2->Outerplanar, 3->Square grid, 4-
>x_tree, 5-> N-vertex D-ary tree,6->debruijn,7->hypercube");
 scanf("%d",&graph);
 switch(graph)
 {
  case 1: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=Complete(E[0],no_of_nodes);
          break;
  case 2: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=Outerplanar(E[0],no_of_nodes);
          break;
  case 3: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=SquareGrid(E[0],no_of_nodes);
          break;
  case 4: printf("\n enter num of nodes");
          scanf("%d",&no_of_nodes);
          no_of_edges=X_tree(E[0],no_of_nodes);
          break;
  case 5: printf("\n enter num of nodes");
```

```
            scanf("%d",&no_of_nodes);
            printf("\n enter depth");
            scanf("%d",&depth);
            no_of_edges=N_D_Tree(E[0],no_of_nodes,depth);
            break;
   case 6: printf("\n enter m and b");
            scanf("%d %d",&m_debruijn,&b);
            no_of_nodes=pow(m_debruijn,b);
            no_of_edges=debruijn(E[0],no_of_nodes,m_debruijn,b);
            break;
   case 7: printf("\n enter dimensions");
            scanf("%d",&dim);
            no_of_nodes=pow(2,dim);
            no_of_edges=hypercube(E[0],dim,no_of_nodes);
            break;
  }
 printf("\n MUTATION OPERATOR:1-> SWAP, 2->INSERT");
 scanf("%d",&mut_choice);
 printf("\n CROSSOVER OPERATOR:1->CX, 2-> PMX");
 scanf("%d",&cross_choice);
 printf("\n METHOD: 1-> 1-D, 2 ->STRAIGHT_LINE_SQUARE, 3 -
>STRAIGHT_LINE_ROOK, 4->HOR_VERT_ROOK");
 scanf("%d",&method_choice);
 printf("\n ENTER pm");
 scanf("%f",&pm);
 printf("\n ENTER pc");
 scanf("%f",&pc);
 if (method_choice==2)
  {
     //k= ceil(sqrt(no_of_nodes));
    printf("\n ENTER K");
    scanf("%d",&k);
  }

 //initialize first individual in population of nodes
 for(i=0;i<no_of_nodes;i++)
 {
   N[0][i]=i;
   Px[0][i]=i;

 }
 j=0;
 for(i=no_of_nodes-1;i>=0;i--)
    Py[0][j++]=i;

 //generate initial population

 InitialPopulation(N,E,Px,Py,no_of_nodes,no_of_edges);
 printf(" pop initialized");
 m=0;
 for(i=0;i<pop_size;i++)
   {
    PageNo[i]=-
    pagenumber(N[i],E[i],Px[i],Py[i],no_of_nodes,no_of_edges)+100;
    PN[i]= -PageNo[i]+100;
   }
 printf(" pagenumber calculated");
```

```
  min_pages=PN[0];
  for(i=1;i<pop_size;i++)
     if(PN[i]<min_pages)
         min_pages=PN[i];


AssignP1P2(N,NextN,E,NextE,Px,NextPx,Py,NextPy,no_of_nodes,no_of_edges)
;
  printf(" pop assigned");
  while(m<max_iterations)
  {


    m=m+1;
    //construct roulette wheel
    tot_pages=0;
    for(i=0;i<pop_size;i++)
        tot_pages=tot_pages+PageNo[i];
    ps[0]=((float)PageNo[0])/(float)tot_pages;
    q[0]=ps[0];
    for(i=1;i<pop_size;i++)
    {
      ps[i]=((float)PageNo[i])/(float)tot_pages;   //compute
                                      probability of selection
      q[i]=q[i-1]+ps[i];
    }
    printf(" roulette construction complete");
    //spin roulette wheel to create new population

    for(i=0;i<pop_size;i++)
    {
     r=realrandom(); //generate real no. between 0 and 1
     if(r<q[0])
       {
        if(method_choice==1||method_choice==2)
            AssignN1N2(NextN[i],N[0],no_of_nodes);// NextN[i]=N[0]
        else
          {
           AssignN1N2(NextPx[i],Px[0],no_of_nodes);
           AssignN1N2(NextPy[i],Py[0],no_of_nodes);
          }
            AssignE1E2(NextE[i],E[0],no_of_edges); //NextE[i]=E[0]
       }
     else
       {
         l=0;
         do
            l+=1;
         while(!(r>=q[l-1] && r<=q[l] || l>pop_size-2));
         if(method_choice==1||method_choice==2)
            AssignN1N2(NextN[i],N[l],no_of_nodes); //NextN[i]=N[l]
         else
          {
           AssignN1N2(NextPx[i],Px[l],no_of_nodes);
           AssignN1N2(NextPy[i],Py[l],no_of_nodes);
          }
```

```
      AssignE1E2(NextE[i],E[l],no_of_edges);
    }
}
printf(" roulette wheel has been spun");
t=-1;
//select individuals for crossover
for(i=0;i<pop_size;i++)
{
  r=realrandom();//generate real no. between 0 and 1
  if(r<pc)
    {
     t=t+1;
     if(method_choice==0||method_choice==1)
        AssignN1N2(c[t],NextN[i],no_of_nodes); //  c[t]=NextN[i]
     else
        {
         AssignN1N2(cx[t],NextPx[i],no_of_nodes);
         AssignN1N2(cy[t],NextPy[i],no_of_nodes);
        }
     index[t]=i;
    }
}

if((t+1)%2!=0)  //odd no of father mother pairs
  {
    t++;
    ran=random(pop_size);
    if(method_choice==0||method_choice==1)
        AssignN1N2(c[t],NextN[ran],no_of_nodes);
       //c[t]=NextN[ran]
     else
        {
         AssignN1N2(cx[t],NextPx[ran],no_of_nodes);
         AssignN1N2(cy[t],NextPy[ran],no_of_nodes);
        }
     index[t]=ran;
  }

printf(" individuals have been selected for crossover");
//select father and mother pairs in sequential order
n=0;i=0;
while(n<=t)
{
  if(method_choice==0||method_choice==1)
   {
     AssignN1N2(father,c[n],no_of_nodes); //father=c[n]
     AssignN1N2(mother,c[n+1],no_of_nodes);// mother,c[n+1]
     n+=2;
     switch(cross_choice)
     {
       case 1:
     crossoverCX(father,mother,child1,child2,no_of_nodes);break;
       case 2:
     crossoverPMX(father,mother,child1,child2,no_of_nodes);break;
     }
     AssignN1N2(NextN[index[i]],child1,no_of_nodes);
  //NextN[index[i]]=child1 ;replace parents by children
```

```
          AssignN1N2(NextN[index[i+1]],child2,no_of_nodes);
          i+=2;
        }
      else
        {
          AssignN1N2(fatherx,cx[n],no_of_nodes);
          AssignN1N2(motherx,cx[n+1],no_of_nodes);
          AssignN1N2(fathery,cy[n],no_of_nodes); //father=c[n]
          AssignN1N2(mothery,cy[n+1],no_of_nodes);
          n+=2;
          crossoverPMX(fatherx,motherx,child1x,child2x,no_of_nodes);
          crossoverPMX(fathery,mothery,child1y,child2y,no_of_nodes);
          AssignN1N2(NextPx[index[i]],child1x,no_of_nodes);   //replace
          parents by children
          AssignN1N2(NextPx[index[i+1]],child2x,no_of_nodes);
          AssignN1N2(NextPy[index[i]],child1y,no_of_nodes);   //replace
          parents by children
          AssignN1N2(NextPy[index[i+1]],child2y,no_of_nodes);
          i+=2;
        }

    }
    printf(" crossover performed");
    //perform mutation
    for(i=0;i<pop_size;i++)
       for(j=0;j<no_of_nodes+no_of_edges;j++)
         {
           r=realrandom();//generates real no. between 0 and 1
           if(r<pm)
              {
                if(j<no_of_nodes)
                  {
                    ran=random(no_of_nodes);  // no. between 0 and
                                              //no_of_nodes-1
                    switch(mut_choice)
                    {
                       case 1: swapN(NextN[i],j,ran);break;
                       case 2: insertN(NextN[i],j,ran);break;
                    }
                  }
                else
                  {
                    ran=random(no_of_edges);
                    swapE(NextE[i],j-no_of_nodes,ran);
                  }
              }
         }
       printf(" mutation performed");

AssignP1P2(N,NextN,E,NextE,Px,NextPx,Py,NextPy,no_of_nodes,no_of_edges)
;  //N=NextN, E=NextE;

    for(i=0;i<pop_size;i++)
      {
       PageNo[i]=-
        pagenumber(N[i],E[i],Px[i],Py[i],no_of_nodes,no_of_edges)+100;
//evaluate population
```

```
        PN[i]= -PageNo[i]+100;
       }

    printf(" second pagenumber computed");
    for(i=0;i<pop_size;i++)
      if(PN[i]<min_pages)
          min_pages=PN[i];

   }//end of max_iterations

printf("\n the minimum pagenumber in all generations was : %d ",
min_pages);

    printf("do you want to quit? 1=yes, 0=no");
    scanf("%d",&choice);

 }while(choice==0);

 getch();

}
```