13. Graph Algorithms

An *algorithm* is a problem-solving method suitable for implementation as a computer program. While designing algorithms we are typically faced with a number of different approaches. For small problems, it hardly matters which approach we use, as long as it is one that solves the problem correctly. However, there are many problems, including some problems in graph theory, for which the known algorithms take so long to compute the solution that they are practically useless. A *polynomial-time algorithm* is one whose number of computational steps is always bounded by a polynomial function of the size of the input. The class of all such problems that have polynomial-time algorithms is denoted by ***P***. For some problems, there are no known polynomial-time algorithms but these problems do have *nondeterministic polynomial-time algorithms*: try all candidates for solutions simultaneously and for each given candidate, verify whether it is a correct solution in polynomial-time. The class of all such problems is denoted by **NP**. Clearly $\mathbf{P} \subseteq \mathbf{NP}$. On the other hand, there are problems that are known to be in ***N*P** and are such that any polynomial-time algorithm for them can be transformed into a polynomial-time algorithm for every problem in **NP**. Such problems are called **NP**-*complete*. Thus, if anybody ever finds a polynomial-time algorithm for an **NP**-complete problem, he or she would have proved that **P= NP**. One of the greatest unresolved problems in mathematics and computer science today is whether $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$ [62].

In this chapter we present algorithms for five well-known graph theory problems. First, we discuss Dijkstra's algorithm for computing shortest paths in graphs. This algorithm is polynomial-time and hence the problem is in **P**. Dijkstra's algorithm is widely used in practice, for example, as part of the TCP/IP protocol suite for routing internet traffic over shortest paths. The second is Prim's algorithm for computing minimal spanning trees in graphs. This algorithm is obtained by a trivial modification of Dijkstra's algorithm, is also polynomial-time and hence the problem again is in **P**. Prim's algorithm is also widely used in practice, for example, in distribution problems and broadcasting data over computer networks. Third, we present Fleury's algorithm for finding Eulerian circuits (cycles) in graphs. The critical point here is to decide whether a certain pair of subgraphs partition the graph into two nontrivial components and the trick is to compute a tree using Prim's algorithm and check whether the tree spans the graph. Fleury's algorithm is also polynomial-time and hence the problem of finding Eulerian circuits is also in **P**. Fourth, we show how to construct the De Bruijn graphs and sequences using Eulerian circuits in polynomial-time. These graphs have recently found important applications in multihop and fault tolerant computer networks. The fifth and final algorithm is for finding Hamiltonian

circuits (cycles) in graphs. The algorithm is an example of a nondeterministic polynomial-time algorithm. The problem of finding a Hamiltonian circuit in a graph is an example of a **NP**-complete problem.

All algorithms are implemented in C++ and tested using Microsoft Visual C++ [158].

13.1 Dijkstra's Algorithm for Shortest Paths

In 1956, Edsger W. Dijkstra [65, 153], was the main programmer at the Burroughs Corporation in Amsterdam, where the construction of one of the earliest computers (the ARMAC) was on the verge of completion. In order to celebrate its inauguration they needed a nice demonstration. It should solve a problem that could be easily stated to a predominantly lay audience. For the purpose of the demonstration, Dijkstra drew a slightly simplified map of the Dutch railroad system; someone in the audience could ask for the shortest connection between, say, Harlingen and Maastricht, and the ARMAC would print out the shortest route town by town. The demonstration was a great success; Dijkstra reminisces that he could show that the inversion of source and destination could influence the computation time required. The speed of the ARMAC and the size of the map were such that one-minute computations always sufficed.

The general problem is to find shortest paths from one specified vertex to all other vertices in a weighted graph, where the edge weights correspond to distances between towns. Together, these paths will form a (not necessarily minimal) spanning tree. The main idea of Dijkstra's algorithm is the following; if P is a shortest path from u to z and P contains v, then the portion of the path P from u to v must be a shortest path from u to v. This suggests that we should determine optimal routes from u to every other vertex z in increasing order of the distance $d(u,z)$ from u to z. We maintain a current tentative distance and use this to update the remaining tentative distances in tabular form. The algorithm works equally well for directed graphs. A formal description of Dijkstra's algorithm is the following.

13.1.1 Dijkstra's algorithm to compute distances from u

Input A weighted graph (or digraph) and a starting vertex u. The weight of edge xy is $w(xy)$; let $w(xy) = \infty$ if xy is not an edge.

Idea Maintain the set S of vertices to which the shortest route from u is known, enlarging S to include all the vertices. To do this maintain also a tentative distance $t(z)$ from u to each z not in S; this is the length of the shortest path found yet from u to z.

Initialisation Let $S = \{u\}$; $d(u,u) = 0$; $t(z) = w(uz)$ for all $z \neq u$.

Iteration Select a vertex v outside S such that $t(v)$ is minimum in the set $\{t(z) | z \notin S\}$. Add v to S. For each edge vz with $z \notin S$, update $t(z)$ to $\min\{t(z), d(u,v) + w(vz)\}$.

Termination Continue the iteration until $S = V(G)$ or until $t(z) = \infty$ for every $z \notin S$. In the first case, all shortest paths from u have been found. In the latter case, the remaining vertices are unreachable from u.

We give an implementation of Dijkstra's algorithm in C++ below.

dijkstra.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <set>

using namespace std;

ifstream infile(``graph.txt'');
ofstream outfile(``shortest_paths.txt'');

int main()
cout<<``Dijkstra's Algorithm.''<<endl;
 int m,n,i,j,k;
 //Read adjacency matrix of weights from graph.txt
 infile>>n;
 vector< vector< float> > weight;
 float val;
 for(i=0; i<n; i++)
 {
  vector< float > row;
  for(j=0; j<n; j++)
  {
   infile>>val;
   row.push_back(val);
  }
  weight.push_back(row);
  }

  //Initialize Table
  const float infinity=1000000;
```

```
vector<bool> known;
for(i=0; i<n; i++) known.push_back(false);
vector<float> d;
d.push_back(0);
for(i=1; i<n; i++) d.push_back(infinity);
vector<int> p;
for(i=0; i<n; i++) p.push_back(-1);
//Print Initial Table
 outfile<<endl<<``INITIAL TABLE:''<<endl;
 outfile<<endl<<``Vertex  : t'';
 for(i=0; i<n; i++) outfile<<i<<'\t';
 outfile<<endl<<``Known   :\t'';
 for(i=0; i<n; i++) outfile<<known[i]<<'\t';
 outfile<<endl<<``Distance:\t'';
 for(i=0; i<n; i++) outfile<<d[i]<<'\t';
 outfile<<endl<<``Path    :\t'';
 for(i=0; i<n; i++) outfile<<p[i]<<'\t';
 outfile<<endl;
//Iteration
for(k=0; k<n; k++)
{
 //Find min of d for unknown vertices
 int min=0;
 while(known[min]==true)min++;
 for(i=0; i<n; i++)
 if(known[i]==false && d[i]<d[min])min=i;
 //Update Table
 known[min]=true;
 for (int j=0; j<n; j++)
 {
  if(weight[min][j]!=0 &&
   d[j]>d[min]+weight[min][j] &&
   known[j]==false)
  {
   d[j]=d[min]+weight[min][j];
   p[j]=min;
  }
```

```
  }
  //Print Table
  outfile<<endl<<``TABLE No.''<<k<<":``<<endl;
  outfile<<endl<<``Vertex  :\t'';
  for(i=0; i<n; i++) outfile<<i<<'\t';
  outfile<<endl<<``Known   :\t'';
  for(i=0; i<n; i++) outfile<<known[i]<<'\t';
  outfile<<endl<<``Distance:\t'';
  for(i=0; i<n; i++) outfile<<d[i]<<'\t';
  outfile<<endl<<``Path    :\t'';
  for(i=0; i<n; i++) outfile<<p[i]<<'\t';
  outfile<<endl;
 }
 //Find shortest paths and spanning tree
 outfile<<endl<<endl<<``SHORTEST PATHS:''<<endl;
 set< vector<int> > span;
 for(i=0; i<n; i++)
{
 outfile<<``Path='';
 vector<int> temp;
 m=i;
 while(m!=-1)
 {
  temp.push_back(m);
  m=p[m];
 }
 outfile<<temp[temp.size()-1]<<" ";
 for(j=temp.size()-2; j>=0; j--)
 {
 outfile<<temp[j]<<" ";
 vector<int> edge;
 edge.push_back(temp[j+1]);
 edge.push_back(temp[j]);
 span.insert(edge);
}
outfile<<``Distance=''<<d[i]<<endl;
}
```

```
//Print spanning tree
outfile<<endl<<‘‘SPANNING TREE:’’<<endl;
set< vector<int>>::iterator it;
for(it=span.begin(); it!=span.end(); it++)
outfile<<‘‘(‘‘<<(*it)[0]<<’’,’’<<(*it)[1]<<")"<<" ";
cout<<‘‘See shortest_paths.txt for results.’’  <<endl;system(‘‘PAUSE’’);
return 0;}
```

Example 13.1.2 Consider the Petersen graph with vertices labelled as shown in Figure 13.1.

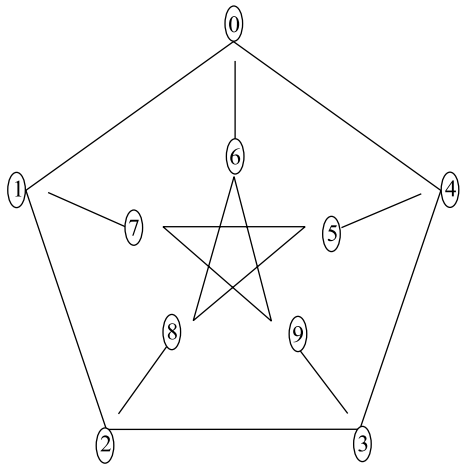

Fig. 13.1 *The Petersen graph with labelled vertices*

Make a text file “graph.txt” as shown below and save it in the same directory as the program “dijkstra.cpp”. The first line of the file “graph.txt” is the number of vertices and the rest is the adjacency matrix of the labelled Petersen graph shown in Figure 13.1.

graph.txt

```
10
0  1  0  0  1  0  1  0  0  0
1  0  1  0  0  0  0  1  0  0
0  1  0  1  0  0  0  0  1  0
0  0  1  0  1  0  0  0  0  1
1  0  0  1  0  1  0  0  0  0
0  0  0  0  1  0  0  1  1  0
1  0  0  0  0  0  0  0  1  1
0  1  0  0  0  1  0  0  0  1
0  0  1  0  0  1  1  0  0  0
0  0  0  1  0  0  1  1  0  0
```

Now compile and run the program "dijkstra.cpp". The following output file "shortest_paths.txt" is produced in the program's directory. The output shows shortest paths from the vertex labelled 0 to all other vertices:

shortest_paths.txt

INITIAL TABLE

Vertex	:	0	1	2	3	4	5	6	7	8	9
known	:	0	0	0	0	0	0	0	0	0	0
Distance	:	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
Path	:	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1

TABLE No.0:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	0	0	0	0	0	0	0	0	0
Distance	:	0	1	∞	∞	1	∞	1	∞	∞	∞
Path	:	−1	0	−1	−1	0	−1	0	−1	−1	−1

TABLE No.1:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	0	0	0	0	0	0	0	0
Distance	:	0	1	2	∞	1	∞	1	2	∞	∞
Path	:	−1	0	1	−1	0	−1	0	1	−1	−1

TABLE No.2:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	0	0	1	0	0	0	0	0
Distance	:	0	1	2	2	1	2	1	2	∞	∞
Path	:	-1	0	1	4	0	4	0	1	-1	-1

TABLE No.3:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	0	0	1	0	1	0	0	0
Distance	:	0	1	2	2	1	2	1	2	2	2
Path	:	-1	0	1	4	0	4	0	1	6	6

TABLE No.4:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	1	0	1	0	1	0	0	0
Distance	:	0	1	2	2	1	2	1	2	2	2
Path	:	-1	0	1	4	0	4	0	1	6	6

TABLE No.5:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	1	1	1	0	1	0	0	0
Distance	:	0	1	2	2	1	2	1	2	2	2
Path	:	-1	0	1	4	0	4	0	1	6	6

TABLE No.6:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	1	1	1	1	1	0	0	0
Distance	:	0	1	2	2	1	2	1	2	2	2
Path	:	-1	0	1	4	0	4	0	1	6	6

TABLE No.7:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	1	1	1	1	1	1	0	0
Distance	:	0	1	2	2	1	2	1	2	2	2
Path	:	−1	0	1	4	0	4	0	1	6	6

TABLE No.8:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	1	1	1	1	1	1	1	0
Distance	:	0	1	2	2	1	2	1	2	2	2
Path	:	−1	0	1	4	0	4	0	1	6	6

TABLE No.9:

Vertex	:	0	1	2	3	4	5	6	7	8	9
Known	:	1	1	1	1	1	1	1	1	1	1
Distance	:	0	1	2	2	1	2	1	2	2	2
Path	:	−1	0	1	4	0	4	0	1	6	6

Shortest paths

Path=0 Distance=0

Path=0 1 Distance=1

Path=0 1 2 Distance=2

Path=0 4 3 Distance=2

Path=0 4 Distance=1

Path=0 4 5 Distance=2

Path=0 6 Distance=1

Path=0 1 7 Distance=2

Path=0 6 8 Distance=2

Path=0 6 9 Distance=2

Spanning tree

(0, 1) (0, 4) (0, 6) (1, 2) (1, 7) (4, 3) (4, 5) (6, 8) (6, 9)

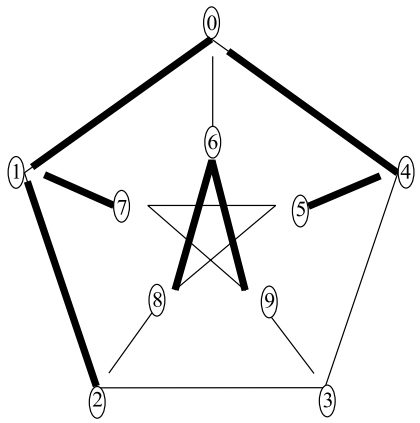

Fig 13.2 *Shortest paths from vertex 0 and a spanning tree*

The output of Dijkstra's algorithm for the Petersen graph with starting vertex 0 is shown in Figure 13.2. In this case, the spanning tree is minimal as the reader may verify. This is not always the case as the next example shows.

Example 13.1.3 Consider the weighted and directed graph shown in Figure 13.3.

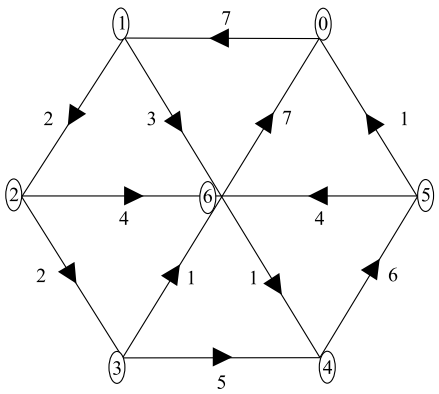

Fig. 13.3 *A weighted and directed graph*

Make a text file "graph.txt" as shown below and save it in the same directory as the program "dijkstra.cpp". The first line of the file "graph.txt" is the number of vertices and the rest is the matrix of weights of the labelled directed graph shown in Figure 13.3 above.

graph.txt

```
7
0 7 0 0 0 0 0
0 0 2 0 0 0 3
0 0 0 2 0 0 4
0 0 0 0 5 0 1
0 0 0 0 0 6 0
1 0 0 0 0 0 4
7 0 0 0 1 0 0
```

Now run the program "dijkstra.cpp". The following output file "shortest_paths.txt" is produced in the program's directory. The output shows shortest paths from the vertex labelled 0 to all other vertices:

shortest_paths.txt

INITIAL TABLE:

Vertex	:	0	1	2	3	4	5	6
Known	:	0	0	0	0	0	0	0
Distance	:	0	∞	∞	∞	∞	∞	∞
Path	:	−1	−1	−1	−1	−1	−1	−1

TABLE No.0:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	0	0	0	0	0	0
Distance	:	0	7	∞	∞	∞	∞	∞
Path	:	−1	0	−1	−1	−1	−1	−1

TABLE No.1:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	0	0	0	0	0
Distance	:	0	7	9	∞	∞	∞	10
Path	:	−1	0	1	−1	−1	−1	1

TABLE No.2:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	1	0	0	0	0
Distance	:	0	7	9	11	∞	∞	10
Path	:	−1	0	1	2	−1	−1	1

TABLE No.3:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	1	0	0	0	1
Distance	:	0	7	9	11	11	∞	10
Path	:	−1	0	1	2	6	−1	1

TABLE No.4:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	1	1	0	0	1
Distance	:	0	7	9	11	11	∞	10
Path	:	−1	0	1	2	6	−1	1

TABLE No.5:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	1	1	1	0	1
Distance	:	0	7	9	11	11	17	10
Path	:	−1	0	1	2	6	4	1

TABLE No.6:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	1	1	1	1	1
Distance	:	0	7	9	11	11	17	10
Path	:	−1	0	1	2	6	4	1

Shortest paths

Path=0 Distance=0

Path=0 1 Distance=7

Path=0 1 2 Distance=9

Path=0 1 2 3 Distance=11

Path=0 1 6 4 Distance=11

Path=0 1 6 4 5 Distance=17

Path=0 1 6 Distance=10

Spanning tree

(0, 1) (1, 2) (1, 6) (2, 3) (4, 5) (6, 4)

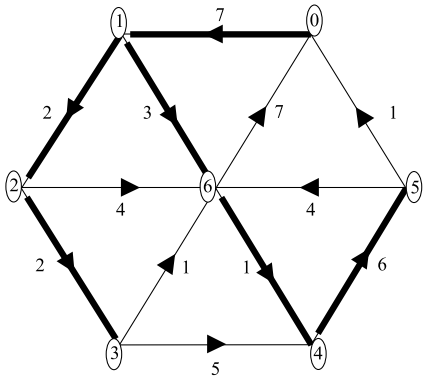

Fig. 13.4 *Shortest paths from vertex 0 and a spanning tree*

The output of Dijkstra's algorithm for the graph of Figure 13.3 with starting vertex 0 is shown in Figure 13.4. In this case, the spanning tree has total weight 21 and is *not* minimal. Indeed, it is possible to find a spanning tree of total weight 19 in this graph, as the next section will show.

13.2 Prim's Algorithm for Minimal Spanning Tree

Almost simultaneously in 1957, while Dijkstra's algorithm was just becoming widely known, R.C. Prim [211, 249] of Bell Labs discovered an algorithm for finding a minimal spanning tree in a weighted graph. Surprisingly, a minor modification of Dijkstra's algorithm allows us to find a minimal spanning tree. A formal description of Prim's algorithm is the following.

13.2.1 Prim's algorithm to compute a minimal spanning tree from u

Input A weighted graph and a starting vertex u. The weight of edge xy is $w(xy)$; let $w(xy) = \infty$ if xy is not an edge.

Idea Maintain the set S of vertices to which the shortest path from u is known, enlarging S to include all the vertices. To do this maintain also a tentative distance $t(z)$ from u to each z not in S; this is the length of the shortest path found yet from u to z.

Initialisation Let $S = \{u\}$; $d(u, u) = 0$; $t(z) = w(uz)$ for all $z \neq u$.

Iteration Select a vertex v outside S such that $t(v)$ is minimum in the set $\{t(z)|z \notin S\}$. Add v to S. For each edge vz with $z \notin S$, update $t(z)$ to $\min\{t(z), w(vz)\}$. (Note this step is different from the corresponding step in Dijkstra's algorithm).

Termination Continue the iteration until $S = V(G)$ or until $t(z) = \infty$ for every $z \notin S$. In the first case, all shortest paths from u have been found. Together, they yield a minimal spanning tree. In the latter case, the remaining vertices are unreachable from u and the shortest paths together will not span the graph.

We give an implementation of Prim's algorithm in C++ below.

prim.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <set>

using namespace std;

ifstream infile(``graph.txt'');
ofstream outfile(``minimal_spanning_tree.txt'');

int main()
{
 cout<<``Prim's Algorithm.''<<endl;
 int m,n,i,j;
 //Read adjacency matrix of weights from graph.txt
 infile>>n;
 vector< vector<float> > weight; {float }val;
 for(i=0; i<n; i++)
 {
  vector<{float}> row;
```

```
 for(j=0; j<n; j++)
 {
  infile>>val;
  row.push_back(val);
 }
 weight.push_back(row);
}
//Initialize Table
const float infinity=1000000;
vector<bool> known;
for(i=0; i<n; i++) known.push_back(false);
vector<float> d;
d.push_back(0);
for(i=1; i<n; i++) d.push_back(infinity);
vector<int> p;
for(i=0; i<n; i++) p.push_back(-1);
//Print Table
 outfile<<endl<<``INITIAL TABLE:''<<endl;
 outfile<<endl<<``Vertex  :\t'';
 for(i=0; i<n; i++) outfile<<i<<'\t';
 outfile<<endl<<``Known   :\t'';
 for(i=0; i<n; i++) outfile<<known[i]<<'\t';
 outfile<<endl<<``Distance:\t'';
 for(i=0; i<n; i++) outfile<<d[i]<<'\t';
 outfile<<endl<<``Path    :\t'';
 for(i=0; i<n; i++) outfile<<p[i]<<'\t';
 outfile<<endl;
//Iteration
for(m=0; m<n; m++)
{
 //Find min of d for unknown vertices
 int min=0;
 while(known[min]==true)min++;
 for(i=0; i<n; i++)
 if(known[i]==false && d[i]<d[min])min=i;
 //Update Table
 known[min]=true;
```

```
   for(j=0; j<n; j++)
   {
    if(weight[min][j]!=0 &&
     d[j]>weight[min][j] &&
     known[j]==false)
    {
     d[j]=weight[min][j];
     p[j]=min;
    }
   }
  //Print Table
  outfile<<endl<<endl<<``TABLE No.''<<m<<``:''<<endl;
  outfile<<endl<<``Vertex  :\t'';
  for(i=0; i<n; i++) outfile<<i<<'\t';
  outfile<<endl<<``Known   :\t'';
  for(i=0; i<n; i++) outfile<<known[i]<<'\t';
  outfile<<endl<<``Distance:\t'';
  for(i=0; i<n; i++) outfile<<d[i]<<'\t';
  outfile<<endl<<``Path    :\t'';
  for(i=0; i<n; i++) outfile<<p[i]<<'\t';
  outfile<<endl;
 }
 //Print minimal spanning tree
 outfile<<endl<<``MINIMAL SPANNING TREE:''<<endl;
 for(i=1; i<n; i++)
 outfile<<``(``<<i<<'',``<<p[i]<<'') '';
 cout<<``See minimal spanning tree.txt.''<<endl;
 system(``PAUSE'');
 return 0;
}
```

Example 13.2.1 Consider the weighted graph shown in Figure 13.5.

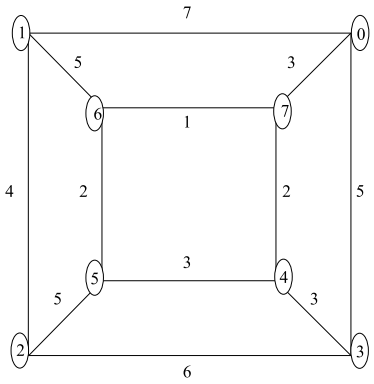

Fig. 13.5 *A weighted graph*

Make a text file "graph.txt" as shown below and save it in the same directory as the program "prim.cpp". The first line of the file "graph.txt" is the number of vertices and the rest is the matrix of weights of the labelled graph shown in Figure 13.5.

graph.txt

```
8
0 7 0 5 0 0 0 3
7 0 4 0 0 0 5 0
0 4 0 6 0 5 0 0
5 0 6 0 3 0 0 0
0 0 0 3 0 3 0 2
0 0 5 0 3 0 2 0
0 5 0 0 0 2 0 1
3 0 0 0 2 0 1 0
```

Now compile and run the program "prim.cpp". The following output file "minimal_spanning_tree.txt" is produced in the program's directory. The output shows a minimal spanning tree for the graph.

minimal_spanning_tree.txt

INITIAL TABLE:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	0	0	0	0	0	0	0	0
Distance	:	∞	∞	∞	∞	∞	∞	∞	∞
Path	:	−1	−1	−1	−1	−1	−1	−1	−1

TABLE No.0:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	1	0	0	0	0	0	0	0
Distance	:	0	7	∞	5	∞	∞	∞	3
Path	:	−1	0	−1	0	−1	−1	−1	0

TABLE No.1:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	1	0	0	0	0	0	0	1
Distance	:	0	7	∞	5	2	∞	1	3
Path	:	−1	0	−1	0	7	−1	7	0

TABLE No.2:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	1	0	0	0	0	0	1	1
Distance	:	0	5	∞	5	2	2	1	3
Path	:	−1	6	−1	0	7	6	7	0

TABLE No.3:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	1	0	0	0	1	0	1	1
Distance	:	0	5	∞	3	2	2	1	3
Path	:	−1	6	−1	4	7	6	7	0

TABLE No.4:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	1	0	0	0	1	1	1	1
Distance	:	0	5	5	3	2	2	1	3
Path	:	−1	6	5	4	7	6	7	0

TABLE No.5:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	1	0	0	1	1	1	1	1
Distance	:	0	5	5	3	2	2	1	3
Path	:	−1	6	5	4	7	6	7	0

TABLE No.6:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	1	1	0	1	1	1	1	1
Distance	:	0	5	4	3	2	2	1	3
Path	:	−1	6	1	4	7	6	7	0

TABLE No.7:

Vertex	:	0	1	2	3	4	5	6	7
Known	:	1	1	1	1	1	1	1	1
Distance	:	0	5	4	3	2	2	1	3
Path	:	−1	6	1	4	7	6	7	0

Minimal spanning tree

[(1, 6) (2, 1) (3, 4) (4, 7) (5, 6) (6, 7) (7, 0)

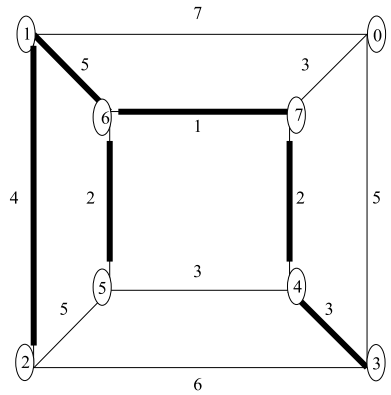

Fig. 13.6 *A minimal spanning tree for the weighted graph*

The output of Prim's algorithm for the graph of Figure 13.5 with starting vertex 0 is shown in Figure 13.6. The spanning tree has total weight 20 and is of minimal total weight compared to all other spanning trees.

Example 13.2.1 It makes sense to try to compute a minimal spanning tree for a directed graph. Consider the weighted directed graph of Figure 13.3. We claimed that the spanning tree found by Dijkstra's algorithm of total weight 21 is *not* minimal. Let us run the program for Prim's algorithm on the file graph.txt of Example 13.1.3. We obtain the following output.

minimal_spanning_tree.txt

INITIAL TABLE:

Vertex :	0	1	2	3	4	5	6
Known :	0	0	0	0	0	0	0
Distance :	0	∞	∞	∞	∞	∞	∞
Path :	−1	−1	−1	−1	−1	−1	−1

TABLE No.0:

Vertex :	0	1	2	3	4	5	6
Known :	1	0	0	0	0	0	0
Distance :	0	7	∞	∞	∞	∞	∞
Path :	−1	0	−1	−1	−1	−1	−1

TABLE No.1:

Vertex :	0	1	2	3	4	5	6
Known :	1	1	0	0	0	0	0
Distance :	0	7	2	∞	∞	∞	3
Path :	−1	0	1	−1	−1	−1	1

TABLE No.2:

Vertex :	0	1	2	3	4	5	6
Known :	1	1	1	0	0	0	0
Distance :	0	7	2	2	∞	∞	3
Path :	−1	0	1	2	−1	−1	1

TABLE No.3:

Vertex :	0	1	2	3	4	5	6
Known :	1	1	1	1	0	0	0
Distance :	0	7	2	2	5	∞	1
Path :	−1	0	1	2	3	−1	3

TABLE No.4:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	1	1	0	0	1
Distance	:	0	7	2	2	1	∞	1
Path	:	−1	0	1	2	6	−1	3

TABLE No.5:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	1	1	1	0	1
Distance	:	0	7	2	2	1	6	1
Path	:	−1	0	1	2	6	4	3

TABLE No.6:

Vertex	:	0	1	2	3	4	5	6
Known	:	1	1	1	1	1	1	1
Distance	:	0	7	2	2	1	6	1
Path	:	−1	0	1	2	6	4	3

Minimal spanning tree

(1, 0) (2, 1) (3, 2) (4, 6) (5, 4) (6, 3)

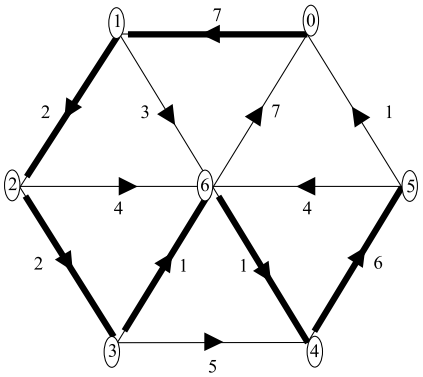

Fig. 13.7 *A minimal spanning tree for the weighted directed graph*

The output of Prim's algorithm for the directed weighted graph of Figure 13.3 with starting vertex 0 is shown in Figure 13.7. The spanning tree has total weight 19 and is of minimal total weight compared to all other spanning trees, in particular, the spanning tree of total weight 21 found by Dijkstra's algorithm.

13.3 Fleury's Algorithm for Eulerian Circuit

The problem of finding an Eulerian circuit in a graph (possibly with multiple edges) has been studied since Leonhard Euler's solution [74] to the problem of the seven bridges of Königsberg in 1736 (see Chapter 3). Lewis Carroll [263], we are told in a biography by his nephew, was fond of asking little children to draw, in one stroke, without lifting the pen off the paper, Figure 13.8.

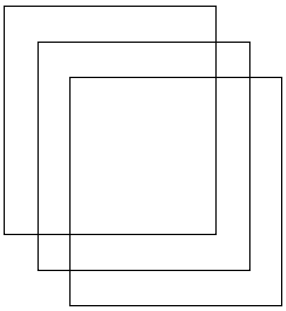

Fig. 13.8 *Lewis Carroll's three-square graph*

If we define all intersections of line segments as vertices then we obtain a planar graph with 18 vertices, known as Lewis Carroll's three-square graph. Drawing the figure in one stroke is equivalent to finding an Eulerian circuit in the graph. Since all vertices have even degree 2 or 4, we know that there *must* be such an Eulerian circuit. In Example 13.3.1, we show how to find this Eulerian circuit in a systematic way. Lucas [153] describes an algorithm for finding an Eulerian trail (or circuit, if one exists) due to Fleury.

13.3.1 Fleury's algorithm

Input A graph G with one nontrivial component and at most two odd vertices.

Initialisation If G has an odd vertex, start at an odd vertex. Else start at any vertex.

Iteration From the current vertex, traverse any remaining edge whose deletion from the remaining graph does not leave a graph with two nontrivial components. To check this, run Prim's algorithm and make sure that the resulting tree spans the graph.

Termination Stop when there are no more edges left to traverse.

We give an implementation of Fleury's algorithm in C++ below. Note that this implementation is for simple graphs, but we can easily modify it to handle graphs with multiple edges.

fleury.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

ifstream infile(``graph.txt'');
ofstream outfile(``eulerian_circuit.txt'');
bool euler(vector<vector<int> > edge);
bool fleury(vector<vector<int> > edge, vector<int> del);
vector<vector<int> >erase(vector<vector<int> > edge, vector<int> del );
bool empty(vector<vector<int> > edge);

int main()
 {
 cout<<``Fleury's Algorithm.''<<endl;
 int n, i, j;
 //Read adjacency matrix of edges from graph.txt
 infile>>n;
 vector<vector<int> > edge; int val;
 for(i=0; i<n; i++)
 {
  vector<int> row;
  for(j=0; j<n; j++)
  {
   infile>>val; row.push_back(val);
  }
  edge.push_back(row);
 }
 cout<<``Read graph from file graph.txt...''<<endl;
 if(euler(edge))
 {
 cout<<``Finding Eulerian circuit...''<<endl;
 vector<int> circuit; int current=0;
```

```
circuit.push_back(current); cout<<current<<'' ";
while(!empty(edge))
{
 for(i=0; i<n; i++)
 {
  int previous=current;
  if(edge[current][i]==1)
  {
   vector<int> del;
   del.push_back(current);
   del.push_back(i);
   if(fleury(edge, del))
   {
    edge=erase(edge,del); current=i;
    circuit.push_back(current);
    cout<<current<<" ";
    break;
   }
  }
 }
}
for(i=0; i<circuit.size(); i++)
outfile<<circuit[i]<<" ";
 cout<<endl<<``See circuit.txt for results.''<<endl;
}
 else
 cout<<``No Eulerian circuit.''<<endl;
 system(``PAUSE''); return 0;
}

bool euler(vector<vector<int> > edge)
{
 for(int i=0; i<edge.size(); i++)
 {
  int deg=0;
  for(int j=0; j<edge[0].size(); j++)
  deg+=edge[i][j];
```

```
   if(deg%2!=0) return false;
  }
  return true;
}
bool fleury(vector<vector<int> > edge, vector<int> del )
{
 int n, i, j, k;
 if(del[0]==del[1]) return false;
 vector<vector<int> > edged=edge;
 edged[del[0]][del[1]]=0; edged[del[1]][del[0]]=0;
 n= edged[0].size();
 //Initialize Table
 const int infinity=1000000;
 vector<bool> known;
 for(i=0; i<n; i++) known.push_back(false);
 vector<int> d; d.push_back(0);
 for(i=1; i<n; i++) d.push_back(infinity);
 vector<int> p;
 for(i=0; i<n; i++) p.push_back(-1);
 //Iteration
 for(k=0; k<n; k++)
 {
  //Find min of d for unknown vertices
  int min=0;
  while(known[min]==true)min++;
  for(i=0; i<n; i++)
  if(known[i]==false && d[i]<d[min])min=i;
  //Update Table
  known[min]=true;
  for(j=0; j<n; j++)
  {
   if(edged[min][j]!=0 && d[j]>edged[min][j] &&
     known[j]==false)

   {
    d[j]=edged[min][j]; p[j]=min;
   }
```

```
   }
  }
  bool ok=true;
  //Find if resulting graph has two nontrivial //components
  it for(i=1; i<n; i++)
  {
   if(p[i]==-1)
   for (int j=0; j<n; j++)
   if(edged[i][j]!=0)
   {ok=false; break;}
  }
  return ok;
 }
 vector<vector<int> > erase(vector< vector<int> > edge, vector<int> del )
 {
  vector< vector<int> > edged=edge;
  edged[del[0]][del[1]]=0; edged[del[1]][del[0]]=0;
  return edged;
 }
 bool empty(vector<vector<int> > edge)
 {
 for(int i=0; i<edge.size(); i++)
 for(int j=0; j<edge[0].size(); j++)
 if(edge[i][j]==1)
 return false; return true;
}
```

Example 13.3.1 Consider the following labelled graph given in Figure 13.9 corresponding to Lewis Carroll's three-square puzzle.

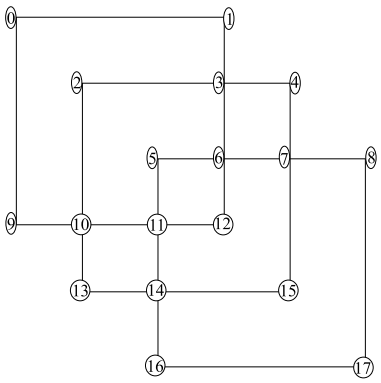

Fig. 13.9 *Labelled graph corresponding to Lewis Carroll's puzzle*

Make a text file "graph.txt" corresponding to Figure 13.9.

graph.txt

```
18
 0  1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
 1  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  1  0  0  0  0  0  0  1  0  0  0  0  0  0  0
 0  1  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0
 0  0  0  1  0  1  0  1  0  0  0  0  1  0  0  0  0  0
 0  0  0  0  1  0  1  0  1  0  0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  1
 1  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  1  0  1  0  1  0  0  0  0
 0  0  0  0  0  1  0  0  0  0  1  0  1  0  1  0  0  0
 0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  1  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  1  0  1  0  1  1  0
 0  0  0  0  0  0  0  1  0  0  0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1
 0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  1  0
```

Now compile and run the program "fleury.cpp". The following output file "eulerian_circuit.txt" is produced in the program's directory.

0 1 3 2 10 11 5 6 3 4 7 6 12 11 14 15 7 8 17 16 14 13 10 9 0 The output shows the final Eulerian circuit in the given graph. If we trace the actual computation of this Eulerian circuit step by step, we obtain a solution to Carroll's puzzle, as shown in Figure 13.10.

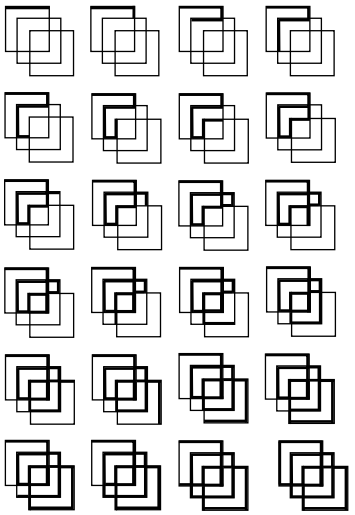

Fig. 13.10 *Solution to Lewis Carroll's three-square puzzle*

13.4 De Bruijn Graphs

To see a good application of Eulerian circuits, we briefly consider the *rotating drum problem*, described as follows. Suppose the head of a rotating drum is divided into 24 = 16 sectors, with each sector labelled with either a zero or a one, as shown in Figure 13.11.

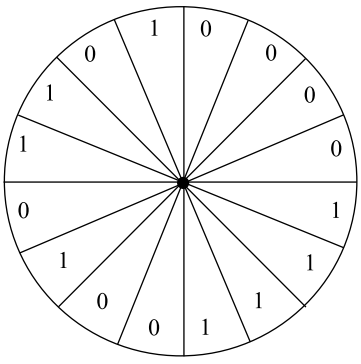

Fig. 13.11 *Labelled rotating drum*

Can the sectors be labelled in such a way that the labels of any four consecutive sectors uniquely determine the position of the drum? This means that the 16 possible quadruples of consecutive binary labels on the drum should be the binary representations of the integers 0 to 15. This question was studied by N.G. de Bruijn [41] in 1946 and thus the resulting binary circular sequences and their corresponding graphs given below are called *De Bruijn sequences* and *De Bruijn graphs*, respectively.

We define a directed graph called G_4 (the De Bruijn graph of order 4) as follows. The vertices are all 3-bit binary strings $x_1x_2x_3$ (each x_i is either zero or one). Thus there are 8 vertices. Each vertex $x_1x_2x_3$ has directed edges to the vertices x_2x_30 and x_2x_31. The directed edge $(x_1x_2x_3, x_2x_3x_4)$ is labelled e_j, where $j = x_1 2^3 + x_2 2^2 + x_3 2^1 + x_4 2^0$ is the unique binary representation of the integer j. Thus, there are 16 directed edges. The graph is shown in the following Figure 13.12.

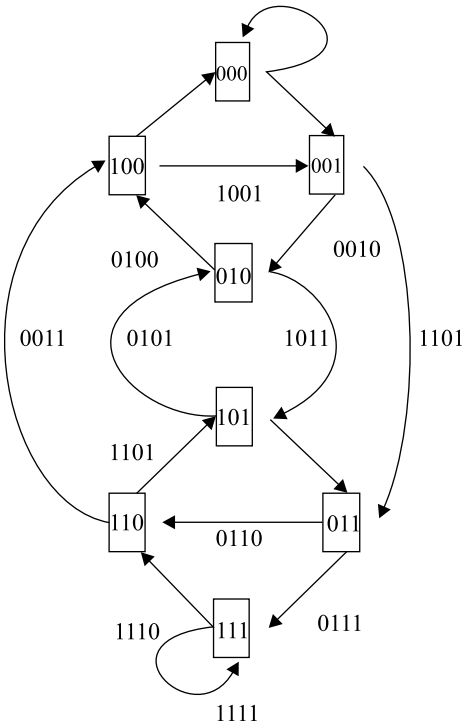

Fig. 13.12 *G_4, the De Bruijn graph of order 4*

Now, G_4 is Eulerian since the in-degree and out-degree of every vertex is 2. Using Fleury's algorithm we find the following Eulerian circuit in G_4: 000, 000, 001, 011, 111, 111, 110, 100, 001, 010, 101, 011, 110, 101, 010, 100, 000. This Eulerian circuit corresponds to the De Bruijn sequence 0000111100101101, to be read circularly, with the required property. This is exactly the labeling on the rotating drum shown in Figure 13.11.

In general, we may build the De Bruijn graph of order n, denoted by G_n, and find circular De Bruijn sequences of length n corresponding to Eulerian circuits in G_n. Among other things, this technique has recently been used to design large scale multihop and fault tolerant computer networks [239].

13.5 Hamiltonian Circuits

A concept that is similar to Eulerian circuits but in reality quite different is that of Hamiltonian circuit. A hamiltonian circuit in a graph is a simple closed path that passes through each vertex exactly once. In the mid 19th century, Sir William Rowan Hamilton [102] tried to popularise the exercise of finding such a circuit in the graph of a dodecahedron. While we have seen a polynomial-time algorithm for Eulerian circuits, no such algorithm is known for Hamiltonian circuits. We give an implementation of a nondeterministic algorithm in C++ below.

hamilton.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

ifstream infile(``graph.txt'');
ofstream outfile(``hamiltonian_circuits.txt'');
int main()
 {
  cout<<``Algorithm for Hamiltonian circuits.''<<endl;
  //Read adjacency matrix of from graph.txt
  int i, j, k, l, m, n;
  infile>>n;
  vector< vector<bool> > graph;
  bool edge;
  for(i=0; i<n; i++)
  {
   vector<bool> row;
```

```
   for(j=0; j<n; j++)
   {
    infile>>edge;
    row.push_back(edge);
   }
   graph.push_back(row);
  }
  int count=0;
  vector<int> vertex;
  for(k=0; k<n; k++)
  vertex.push_back(k);
  cout<<``\nStarting search...\n'';
  bool found=false, circuit=true;
while(next_permutation(vertex.begin(),vertex.end())
)
 {
  if(vertex[0]!=0) break;
  for(l=0; l<n; l++)
   circuit=
     circuit*graph[vertex[l%n]][vertex[(l+1)%n]];
  switch(circuit)
  {
   case true:
    found=true;
    count++;
    cout<<endl
      <<count
      <<`` Hamiltonian Circuits found:''<<endl;
    outfile<<endl
           <<count
           <<`` Hamiltonian Circuits found:''
           <<endl;
    for(m=0; m<n; m++)
    {
     cout<<vertex[m]<<`` '';
     outfile<<vertex[m]<<`` '';
    }
```

```
    cout<<endl; outfile<<endl;
    break;
    default:
    break;
   }
  circuit=true;
 }
 if(!found)
 {
  cout<<''\nNo Hamiltonian Circuits found.\n'';
  outfile<<''\nNo Hamiltonian Circuits found.\n'';
 }
 cout<<''\nSee hamilton_circuits.txt.''
     <<endl;
 system(''PAUSE'');
 return 0;
}
```

Example 13.5.1 We consider the graphs of the five platonic solids, all of which are known to be Hamiltonian. First, consider the simplest of the platonic graphs, the tetrahedron:

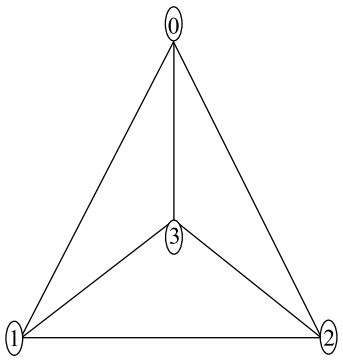

Fig 13.13 *Labelled graph of the tetrahedron*

Make a text file "graph.txt" as shown below and save it in the same directory as the program "hamilton.cpp". The first line of the file "graph.txt" is the number of vertices and the rest is the adjacency matrix of the labelled tetrahedron graph shown in Figure 13.13.

graph.txt

```
4
0  1  1  1
1  0  1  1
1  1  0  1
1  1  1  0
```

Now compile and run the program. We find five Hamiltonian circuits: 0 1 3 2, 0 2 1 3, 0 2 3 1, 0 3 1 2, 0 3 2 1. Next, consider the graph of the octahedron:

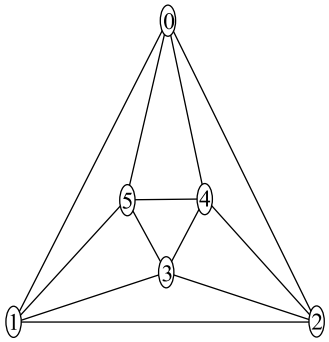

Fig. 13.14 *Labelled graph of the octahedron*

The file "graph.txt" for the octahedron graph is shown below:

graph.txt

```
6
0  1  1  0  1  1
1  0  1  1  0  1
1  1  0  1  1  0
0  1  1  0  1  1
1  0  1  1  0  1
1  1  0  1  1  0
```

Now run the program. We find 31 Hamiltonian circuits: 0 1 2 3 5 4, 0 1 2 4 3 5, 0 1 3 2 4 5, 0 1 3 5 4 2, 0 1 5 3 2 4, 0 1 5 3 4 2, 0 1 5 4 3 2, 0 2 1 3 4 5, 0 2 1 3 5 4, 0 2 1 5 3 4, 0 2 3 1 5 4, 0 2 3 4 5 1, 0 2 4 3 1 5, 0 2 4 3 5 1, 0 2 4 5 3 1, 0 4 2 1 3 5, 0 4 2 3 1 5, 0 4 2 3 5 1, 0 4 3 2 1 5, 0 4 3 5 1 2, 0 4 5 1 3 2, 0 4 5 3 1 2, 0 4 5 3 2 1, 0 5 1 2 3 4, 0 5 1 3 2 4, 0 5 1 3 4 2, 0 5 3 1 2 4, 0 5 3 4 2 1, 0 5 4 2 3 1, 0 5 4 3 1 2, 0 5 4 3 2 1.

Next, consider the graph of the cube.

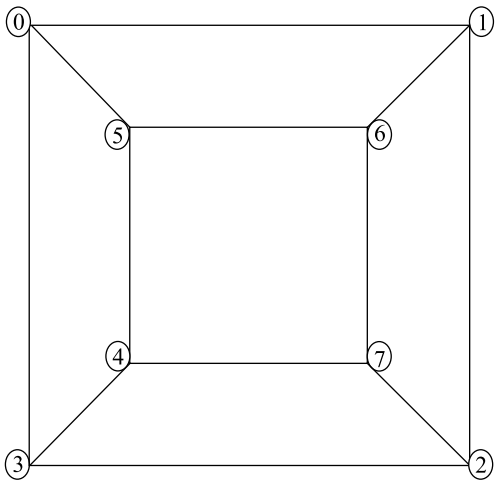

Fig. 13.15 *Labelled graph of the cube*

graph.txt

```
8
0 1 0 1 0 1 0 0
1 0 1 0 0 0 1 0
0 1 0 1 0 0 0 1
1 0 1 0 1 0 0 0
0 0 0 1 0 1 0 1
1 0 0 0 1 0 1 0
0 1 0 0 0 1 0 1
0 0 1 0 1 0 1 0
```

Now run the program. We find 12 Hamiltonian circuits: 0 1 2 3 4 7 6 5, 0 1 2 7 6 5 4 3, 0 1 6 5 4 7 2 3, 0 1 6 7 2 3 4 5, 0 3 2 1 6 7 4 5, 0 3 2 7 4 5 6 1, 0 3 4 5 6 7 2 1, 0 3 4 7 2 1 6 5, 0 5 4 3 2 7 6 1, 0 5 4 7 6 1 2 3, 0 5 6 1 2 7 4 3, 0 5 6 7 4 3 2 1. Next, consider the graph of the icosahedrons, given in Figure 13.16.

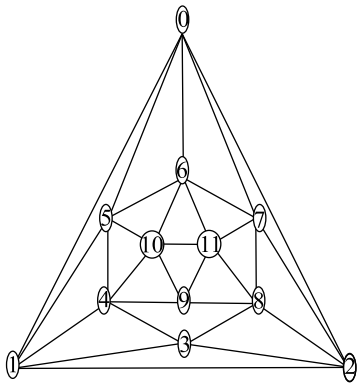

Fig. 13.16 *Labelled graph of the icosahedron*

graph.txt

```
12
 0  1  1  0  0  1  1  1  0  0  0  0
 1  0  1  1  1  1  0  0  0  0  0  0
 1  1  0  1  0  0  0  1  1  0  0  0
 0  1  1  0  1  0  0  0  1  1  0  0
 0  1  0  1  0  1  0  0  0  1  1  0
 1  1  0  0  1  0  1  0  0  0  1  0
 1  0  0  0  0  1  0  1  0  0  1  1
 1  0  1  0  0  0  1  0  1  0  0  1
 0  0  1  1  0  0  0  1  0  1  0  1
 0  0  0  1  1  0  0  0  1  0  1  1
 0  0  0  0  1  1  1  0  0  1  0  1
 0  0  0  0  0  0  1  1  1  1  1  0
```

Now, it gets interesting. The program runs for quite a few minutes! We find 2560 Hamiltonian circuits: 0 1 2 3 4 5 6 10 9 8 11 7, ..., 0 7 11 10 9 8 3 2 1 4 5 6. Finally, consider the graph of the dodecahedron, the original inspiration for Hamilton [212], given in Figure 13.17.

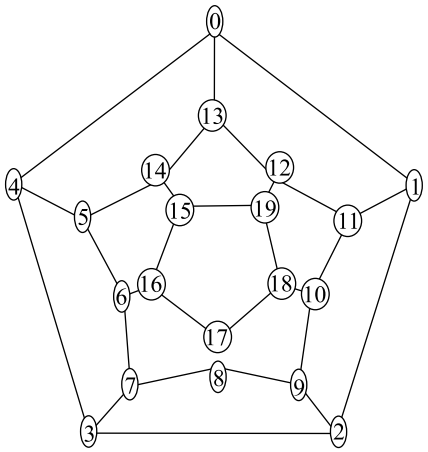

Fig. 13.17 *Labelled graph of the dodecahedron*

The file "graph.txt" for the dodecahedron graph is shown below:

graph.txt

```
20
 0  1  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  0  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  1  0  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  1  0  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0
 1  0  0  1  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0
 1  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0
 0  1  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0
 0  0  0  1  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0
 0  0  0  0  1  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0
 0  0  0  0  0  1  0  0  0  1  0  0  0  0  0  0  0  0  0  1
 0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  1  0  0  0  0
 0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  1  0
 0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  1
 0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1  0  1  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1  0  1  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1  0  1
 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  1  0
```

Depending on the speed of your computer, the program will now run for quite a few hours, and perhaps even days. One of the Hamiltonian circuits we found was: 0 1 2 3 4 5 6 7 8 9 10 11 12 19 18 17 16 15 14 13.