7 Graph Representations and Traversals

7.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing *graphs*.
- To get hands-on experience with implementing graphs, and some graphs processing algorithms.
- To develop critical thinking concerning implementation decisions for graphs, and some graphs processing algorithms.

The outcomes for this session are:

- Improved skills in C programming using pointers and dynamic memory allocation.
- A clear understanding of graphs, the operations they support, and some of their use.
- The ability to make implementation decisions concerning graphs.
- The ability to decide when graphs are suitable in solving a problem.

7.2 Brief Theory Reminder

Basic Notions

A *directed graph* (*digraph*, for short) $G = (V, E)$ is an ordered pair of elements of two sets: V is a set of *vertices* (or nodes) and $E : V \times V$ is a set of arcs.

An *arc* (also called an *edge*) is an ordered pair of vertices (v, w); v is called the *tail* and w the *head* of the arc. We say that arc $v \to w$ is *from v to w*, and that s*w is adjacent to v*.

A *path* in a digraph is a sequence of vertices $v_1, v_2, ..., v_n$ such that $v_1 \to v_2$, $v_2 \to v_3$, ..., $v_{n-1} \to v_n$, are arcs, i.e $(v_i, v_{i+1}) \in E$. This path is *from* vertex v_1 to vertex v_n and *passes through* vertices $v_2, v_3, \ldots, v_{n-1}$, and ends at vertex v_n. There is a path of length 0 from any vertex to itself.

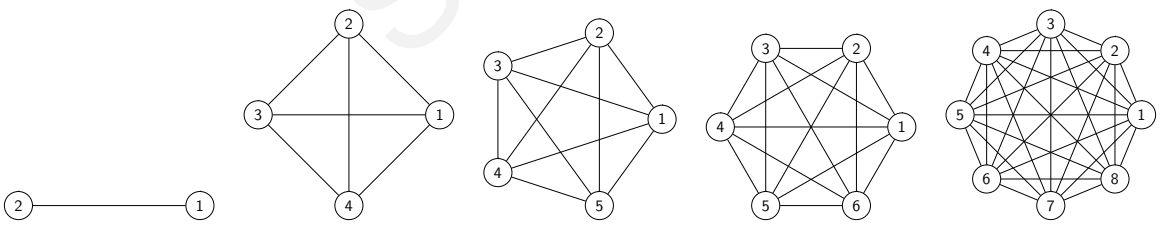
A path is *simple* if all vertices on the path, except possibly the first and the last, are distinct.

A simple *cycle* is a simple path of length at least one that begins and ends at the same vertex.

A *labeled* directed graph is a digraph where every arc and/or vertex has a label associated with it. The label may be a name, a cost or an arbitrary value.

A digraph is *strongly connected* if for any two vertices v and w there is a path from v to w (and, of course, one from w to v).

A *complete* graph is a graph in which each pair of graph vertices is connected by an edge. The complete graph with n graph vertices is denoted K_n and has $binonn2 = n(n-1)/2$ (the triangular numbers) undirected edges, where $binomnk$ is a binomial coefficient. In older literature, complete graphs are sometimes called universal graphs. Figure 7.1 shows complete graphs K_2, K_4, K_5, K_6, and K_8.

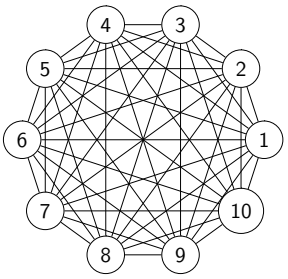


(a) K_2. (b) K_4. (c) K_5. (d) K_6. (e) K_8.

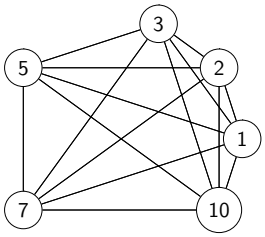
Figure 7.1: Complete graphs examples.

A *subgraph* of a graph G is another graph $G' = (V', E')$ where $V' \subset V$ and $E' \subset E$. A *vertex-induced* subgraph (sometimes simply called an "induced subgraph") is a subset of the vertices of a graph G together with any edges whose endpoints are both in this subset (formally, $G' = (V', E \cap (V' \times V'))$). Figure 7.2 shows an example.

An *undirected graph* (or simply a *graph*) $G = (V, E)$ is a pair of a set of vertices, V, and a set of arcs (or *edges*), E. An edge is an unordered pair $(v, w) = (w, v)$ of nodes. Definitions given for digraphs also hold for graphs.



(a) The complete graph K_{10}.



(b) The subgraph induced by $V' = \{1, 2, 3, 5, 7, 10\}$ subset of K_{10}.

Figure 7.2: Complete graph K_{10} and an induced subgraph of it.

Representation Methods

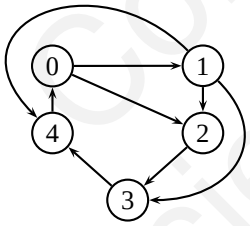
Both digraph and undirected graphs are usually represented using *adjacency matrices* and *adjacency lists*.
For a digraph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, the adjacency matrix is defined as:

$$A[i][j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$$

A *labeled* adjacency matrix, A, also called a *cost* matrix is defined as

$$A[i][j] = \begin{cases} \text{label of arc } (i,j) & \text{if } (i,j) \in E \\ \text{an arbitrary symbol} & \text{if } (i,j) \notin E \end{cases}$$

The adjacency matrix is symmetric for undirected graphs.
In an adjacency list the list of arcs for adjacent nodes is kept. The whole graph may be represented by a table indexed by nodes, each entry for a node i in the table containing the address of the list of nodes adjacent to i. That list may be a static or a dynamic list. Figure 7.3 shows a graph and its adjacency matrix representation, and Figure 7.4 shows static and dynamic adjacency lists representations.



(a) A digraph.

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	1	1
2	0	0	0	1	0
3	0	0	0	0	1
4	1	0	0	0	0

(b) Adjancency matrix representation for the digraph of Figure 7.3(a)

Figure 7.3: A digraph and its adjacency matrix representation

Which of the two representations is best?

The adjacency *matrix* is useful when the presence or absence of an arc is frequently tested. It performs badly when the number of arcs is much less than the square of the number of nodes (i.e. $|E| \ll |V|^2$).

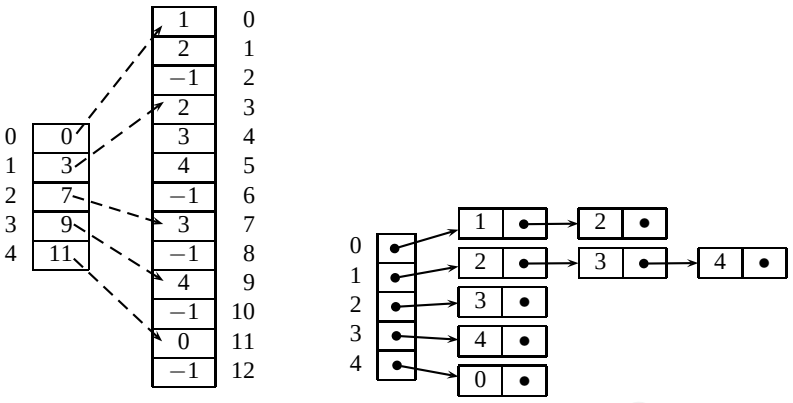
The adjacency *list* makes better use of memory, but looking for arcs is more computationally intensive.

Traversals

Breadth-first Search

Breadth first search involves the following actions:

1. Enqueue the start vertex in an empty queue.



(a) Static adjacency list (b) Dynamic adjacency list.

Figure 7.4: Adjacency list representations for the graph of Figure 7.3(a)

2. Dequeue one node to process and enqueue all its unvisited adjacent nodes.
3. Repeat step 2 until the queue becomes empty.

A sketch of the algorithm is shown in Listing 7.1.

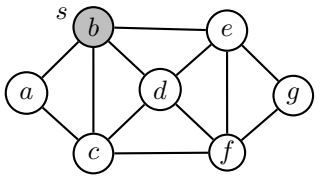
Listing 7.1: A breadth-first search implementation for a graph

```
enum { UNVISITED, VISITED };
void bfs(int nbOfNodes, int srcNode)
{
    int mark[nbOfNodes]; /* for marking visited nodes */
     int w; /* node */

    QueueT Q = createEmptyQueue(); /* queue of nodes - integers */
    for (int i = 0; i < nbOfNodes; i++) /* mark all nodes unvisited */
        mark[i] = UNVISITED;
    mark[srcNode] = VISITED; /* mark source node visited */
    process info for srcNode;
    enqueue(Q, srcNode);
    /* srcNode will be the first node dequeued in the loop below */
    while(! empty(Q))
    {
        int v = dequeue(Q);
        for (each node w adjacent to v)
            if (mark[w] == UNVISITED)
            {
                mark[w] = VISITED;
                process info for w;
                enqueue(Q, w);
            }
    }
}
```

A trace of the relevant steps in breadth-first search is shown in Figure 7.5.

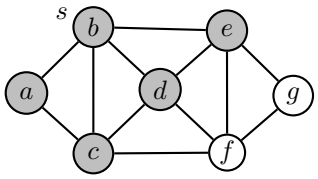
Depth-first Search

When searching a graph in *depth-first* search mode, the source node is marked *visited* first. After visiting all the nodes reachable from given source, the traversal ends. If there still are unvisited nodes, another node is chosen as a source an the steps are repeated. The algorithm sketch is shown in Listing 7.2.



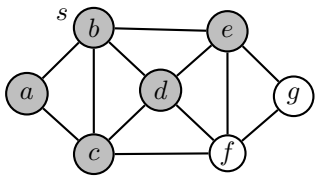
$Q : b$

(a) Before entering the while-loop.



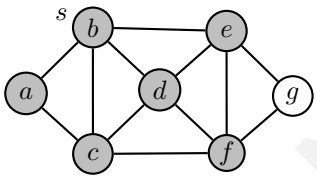
$Q : a, c, d, e$

(b) After processing nodes adjacent to b.



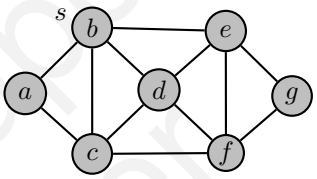
$Q : c, d, e$

(c) After processing nodes adjacent to a.



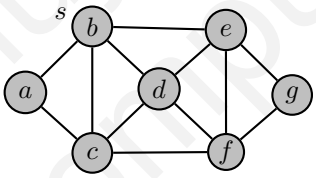
$Q : d, e, f$

(d) After processing nodes adjacent to c.



$Q : f, g$

(e) After processing nodes adjacent to d and e.



Q : empty

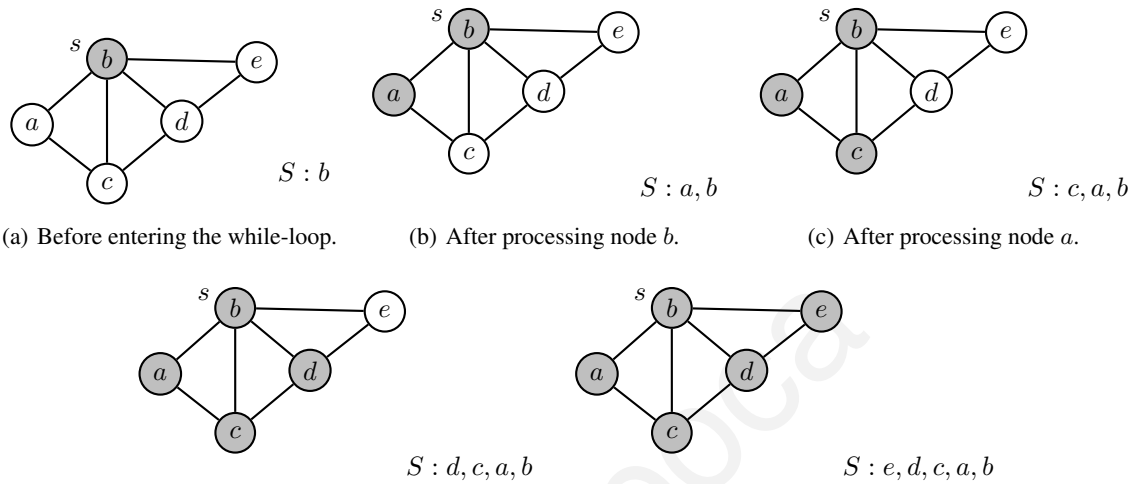
(f) After processing nodes adjacent to f and g.

Figure 7.5: A trace of breadth-first search on a graph.

Listing 7.2: A depth-first search implementation for a graph

```
enum { UNVISITED, VISITED };
void dfs(int nbOfNodes, int srcNode)
{
    int mark[nbOfNodes]; /* for marking visited nodes */
    int w; /* node */

    StackT S = createEmptyStack(); /* stack of nodes */

    for (int i = 0; i < nbOfNodes; i++) /* mark source node visited */
        mark[i] = UNVISITED;
    mark[srcNode] = VISITED; /* mark source node visited */

    /* process info for srcNode; */
    process(srcNode);

    push(S, srcNode);
    while (!empty(S))
    {
        int v = top(S);
        let w be the next unvisited node on AdjList(v);
        if (w exists)
        {
            mark[w] = VISITED;
            process info for w;
            push(S, w);
        }
        else
            pop(S);
}
```

Note that this algorithm uses a stack to eliminate recursion. A trace of the relevant steps in depth-first search is shown in Figure 7.6.



(a) Before entering the while-loop. (b) After processing node b. (c) After processing node a.

(d) After processing node c. (e) After processing node d.

Figure 7.6: A trace of depth-first search on a graph.

Finding Shortest (Minimum Cost) Paths from a Single Vertex

Let $G = (V, E)$ be a labeled directed graph, where every arc is labeled with a non-negative number called a *cost*. The graph may be represented using the labeled adjacency matrix (also called a cost matrix).
Given two vertices, one denoted as the source and the other as the destination, one problem is to find the minimum cost path from the source to the destination. Dijkstra's solution to this problem is characterized by the following:

- A set S of vertices $j \in V$, for which there is at least one path from the source vertex, say s, to the destination, j, is maintained. Initially $S = \{s\}$.
- At each step, a vertex v whose distance to a vertex $w \in S$ is minimal, is added to S.

In order to record the minimal paths from the source s to each other vertex, we will use an array $parent$, in which $parent[k]$ holds the vertex preceding k on the shortest path.
We use the following notations in describing Dijkstra's algorithm:

- n is the number of vertices in V, i.e. $n = |V|$.
- The set S is represented by its characteristic vector, i.e. $S[i] = 1$ if $i \in S$, and $S[i] = 0$ if $i \notin S$.
- An $n \times n$ matrix, $cost$, defined as follows:

$$cost : \begin{cases} cost[i,j] = c,\ c > 0 & \text{if } (i,j) \in E \\ cost[i,j] = 0 & \text{if } i = j \\ cost[i,j] = \infty & \text{if } (i,j) \notin E \end{cases}$$

- The vector $parent$ holds the vertices which are accessible from the source vertices. It allows for path reconstruction – from a source to any accessible node.
- For nodes which cannot be reached from a source node, $S[i] = 0$ and $dist[i] = \infty$.

A implementation of Dijkstra's algorithm is presented in Listing 7.3.

Listing 7.3: An implementation fo Dijkstra's algorithm.

```
#define NMAX ? /* max no. of nodes */
#define INFTY ? /* big value for infinity */
double dist[NMAX]; /* distances */
double cost[NMAX][NMAX];
int parent[NMAX];
int S[NMAX];

/* nbOfNodes = number of nodes in the graph
```

```
   source = source node id */
void Dijkstra(int nbOfNodes, int source)
{
    int k;
    /* initialize */
    for (int i = 1; i <= n; i++)
    {
        S[i] = 0; /* set S empty */
        dist[i] = cost[source][i];
        if (dist[i] < INFTY)     parent[i] = source;
        else                     parent[i] = 0;
    }
    /* add source to set S */
    S[source] = 1;
    parent[source] = 0;
    dist[source] = 0;
    /* build vectors dist and parent */
    for (int step = 1; step <= n-1; step++)
    {
        find unselected vertex k with dist[k] minimal;
        if (minimum found == INFTY) return;
        S[k] = 1; /* add k to set S */
        for (int j = 1; j <= n; j++)
            if (S[j] == 0 && dist[k] + cost[k][j] < dist[j])
            {
                dist[j] = dist[k] + cost[k][j];
                parent[j] = k;
            }
    }
}
```

All Pairs Shortest Paths

By repeatedly applying Dijkstra's algorithm with each node as a source, in turn, we can obtain the minimum paths for all the pairs of vertices in a graph. Another choice is to use R. W. Floyd's algorithm. Floyd's algorithm was developed for solving the all pairs shortest paths problem.
The algorithm maintains the minimal costs in an array, say A. Initially, A is identical to the cost matrix, $cost$. The minimal distances computation is accomplished in n iterations, where n is the number of nodes. At iteration k, $A[i][j]$ holds the minimum distance between nodes i and j using paths which do not contain nodes numbered higher than k, except possibly for the end nodes, i and j. A is calculated with the following formula:

$$A_{ij}{}^{(k)} = \min(A_{ij}{}^{(k-1)}, A_{ik}{}^{(k-1)} + A_{kj}{}^{(k-1)})$$

Because $A_{ik}{}^{(k)} = A_{ik}{}^{(k-1)}$ and $A_{kj}{}^{(k)} = A_{kj}{}^{(k-1)}$ we may use a single copy of array A. Floyd's algorithm may be implemented as shown in Listing 7.4.

Listing 7.4: An implementation of Floyd's algorithm for all pairs shortest paths problem.

```
#define NMAX ? /* max no. of nodes */
double cost[NMAX][NMAX];
double A[NMAX][NMAX];

void Floyd(int nbOfNodes)
{
    /* initialize A */
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = cost[i][j];
    for (int i = 0; i < n; i++)
        A[i][i] = 0;
    for (int k = 0; k < n; k++)      /* all nodes */
        for (int i = 0; i < n; i++) /* all rows */
            for (int j = 0; j < n; j++) /* all columns */
                if (A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j];
}
```

For keeping track of the shortest paths, we can make use of an additional table, p, where $p[i, j]$ will hold the vertex k which lead to the minimum distance $A[i, j]$. If $p[i, j] = 0$, then the edge (i, j) is the shortest from i to j. In order to display the intermediate vertices along the path from i to j, we can use the function in Listing 7.5.

Listing 7.5: Listing the vertices along the path.

```
void path(int i, int j)
{
    int k;
    k = p[i][j];
    if (k != 0)
    {
        path(i, k);
        list node k;
        path(k, j);
    }
}
```

7.3 Laboratory Assignments

Mandatory Assignments

Implement modular C programs which have input and output stored in files with names given as command line arguments to solve the following exercises:

7.3.1. Let G be a digraph, i.e. $G = (V, E)$ and let V' be one of its subgraphs. If the nodes are represented by integers which are to be read from the standard input, show the induced subgraph $G' = (V', E')$. Input is given as follows:

```
V␣nodes␣1␣3␣5␣6␣7
V␣arcs␣(1␣3)␣(1␣5)␣(1␣7)␣(3␣6)␣(3␣7)␣...
V'␣nodes␣1␣3␣5
```

Output should look like this:

```
V'␣arcs␣(1␣3)␣(1␣5)
```

Hint. Use *fgets* and *sscanf* to read input. E.g. reading a pair of arcs can be accomplished with `sscanf(&buffer[i], "(%d%d)", &v, &w)` where v and w are integers.

7.3.2. Implement breadth-first and depth-first traversals for a graph represented by an adjacency matrix. Use the algorithms of §7.2. Input should be as in the lines `V nodes` and `V arcs` of exercise 7.3.1.

7.3.3. Implement Dijkstra's algorithm for minimal cost paths in a directed graph. .
I/O description. Input consists of pairs of nodes given as numbers, separated by a comma and followed by an equal sign followed by the cost attached to that edge $n_1,\ n_2 = n_3$ which are connected by an edge. **E.g.**

```
1,2=22
1,3=11
1,5=5
1,7=5
2,3=55
2,4=41
2,7=33
3,4=8
3,5=4
3,6=9
4,6=17
4,7=9
5,7=20
```

The output should look like this (this an unchecked example):

```
99 1-7-5-3-6-4-2 (5,20,4,9,17,41)
```

where the first number is the minimum cost, followed by the list node identifications and then by the list of arc lengths.

Optional Assignments

7.3.4. Implement graph construction for a graph $G = (V, E)$ for the three representation methods presented at §7.2. Also write a function to print the graphs. Format for input and output should be the same as in exercise 7.3.1.

7.3.5. Implement a function which should check if there is a path between two given nodes, say v and w, of a directed graph $G = (V, E)$. Graphs will be read from files as in exercise 7.3.1, and paths will be printed as a sequence of nodes (i.e integer numbers) separated by a single space.

7.3.6. For an undirected graph $G = (V, E)$, implement a function to check whether or not the graph is strongly connected. Graphs will be read from files input as in exercise 7.3.1, output $\in \{"yes", "no"\}$.

7.3.7. Implement Floyd's algorithm. Use the same format for input and output as in the exercise 7.3.3. What can you tell about the running time of this algorithm?