CHAPTER 4: GRAPHS

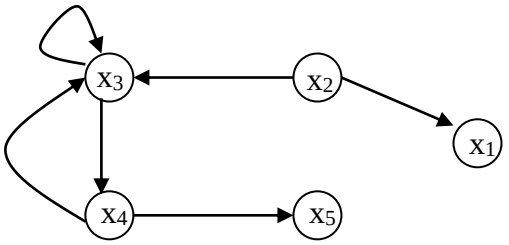
Contents

1. Introduction to graphs

Graphs are currently the preferred tool for modeling sets of complex structures. They are essential for representing and studying relationships between objects.

Graphs are used in:

- Economy (organizational diagram)
- Electronics (integrated circuits)
- Databases (relations between information)
- Communications (networks)
- Transport (road networks)
- Scheduling (project structure)
- Etc.



2. Definitions

2.1 Directed graphs

- A graph $\boldsymbol{G}$ is a couple $(\boldsymbol{X}, \boldsymbol{U})$ where:
  - X is a set $\{x_1, \dots, x_n\}$ of ***nodes or vertices***.
  - $U = \{u_1, u_2, \dots, u_m\}$ is a set of ordered pairs of nodes called **arcs**.

- A graph is **weighted** if $\exists$ an application $C : U \rightarrow R$, associating to each arc u a real c_u.

- A **loop** is an arc connecting a node to itself.

- Each arc $u = (x, y)$ have two ends called **initial** (x) and **terminal** (y). u is said **internally incident** to x and **externally incident** to y.

- In an arc of the form (x, y), x is called **predecessor** of y and y is called **successor** of x. x and y are called neighbors (or adjacent).

- The set of successors of x is noted $\Gamma(x)$ and that of its predecessors is noted $\Gamma^{-}(x)$.

- The number of successors of x is called **exterior half degree** and is noted $d_G^+(x) = |\Gamma(x)|$. The **interior half degree** of x is defined as $d_G^-(x) = |\Gamma^{-}(x)|$. The **degree** of x is defined as follows: $d_G(x) = d_G^+(x) + d_G^-(x)$.

- The graph **density** is defined by the ratio $\boldsymbol{m/n^2}$ i.e. the current number of arcs of G divided by the maximum number of arcs the graph G can have. Most graphs encountered in practice are not very dense (low density). They are called sparse.

2.2 Undirected graphs

In applications where we are only interested in the pairs of connected nodes and not in their orientations, we use undirected graphs. In this type of graphs, we use the term edge $e = [x, y]$ rather than arc (x, y). An undirected graph is noted $G = (X, E)$ where E denotes the set of edges. Terms that apply to directed graphs and are independent of orientation remain valid for undirected graphs.

2.3 Exploring graphs

- A **path** μ of length p is a sequence of p arcs $(u_1, \dots, u_p)$ such as the initial node of u_i is equal to the terminal node of u_{i-1} $\forall\, i > 1$, and the terminal node of u_i is equal to the initial node of u_{i+1} $\forall\, i < p$.

- A **circuit** is a closed path, that is a path such as $u_1 = u_p$.

- An arc is a path of length 1 and a loop is a circuit of length 1 too.

- In the case of undirected graphs, we use the term **chain** rather than path, and **cycle** rather than circuit.

- A graph **traversal** is an element of the set of **paths**, **circuits**, **chains** and **cycles**.

2.4 Particular graphs

- A directed graph $G = (X, U)$ is **symmetric** if $(x, y) \in U \Longrightarrow (y, x) \in U$. Undirected graphs can be represented by symmetric oriented graphs.

- The **complementary** graph of $G = (X, U)$ is the graph $H = (X, X^2 - U)$.

- A graph is **complete** if all pairs of nodes are connected by an arc or an edge.

3. Graph representation

3.1 Adjacency matrix

Consider a directed graph $G = (X, U)$ whose number of nodes is n and that of arcs is m. An adjacency matrix is a matrix $M\ (n \times n)$ whose elements are Boolean indicating whether corresponding nodes are connected or not.

A weighted graph $G = (X, U, W)$ can be represented by a matrix $W\ (n \times n)$ which gives both the weight of the arcs and their existence. With adjacency matrices we can easily detect loops, symmetry and connectivity. Using this data structure, we can also easily obtain the list of successors of a node as well as the list of its predecessors. However, this structure are not adequate for graphs with low density.

Example: The graph given in section 2.1. is represented by the following adjacency matrix M:

M	x_1	x_2	x_3	x_4	x_5
x_1	0	0	0	0	0
x_2	1	0	1	0	0
x_3	0	0	1	1	0
x_4	0	0	1	0	1
x_5	0	0	0	0	0

3.2 Adjacency lists

a. Contiguous lists

Adjacency lists implement the structure $G = (X, \Gamma)$ where successors are stored consecutively in arrays or in linked lists. In the case of arrays, we denote by **Succ** the array whose elements are the lists of the successors of the nodes $0,1, \dots, n-1$, i.e $\Gamma(0), \Gamma(1), \dots, \Gamma(n-1)$. The size of array **Succ** is thus the number of arcs i.e. m. To delimit these successive lists of successors, we use an array called **Head** of size n, which gives for each node the index in the array **Succ** where its successors begin. Indeed, the successors of a node are arranged in **Succ** from **Head[y]** to **Head[y+1]-1**. In the case where a node y has no successors, we set **Head[y]=Head[y+1]**.

Undirected graphs are represented as symmetric directed graphs. If $[x, y]$ is an edge, x appear in the list of successors of y, and y in that of x. In the case of a weighted graph $G = (X, U, W)$, we use another array **W** which contains weights that correspond to successors stored in array **Succ**.

The major advantage of this representation is its compactness. In fact, the space complexity is $n + m + 1 = O(m)$ which is much less than n^2 (for adjacency matrix) particularly if the graph G is not sparse.

Example: (Graph section 2.1)

1	2	3	4	5	6
1	1	3	5	7	7

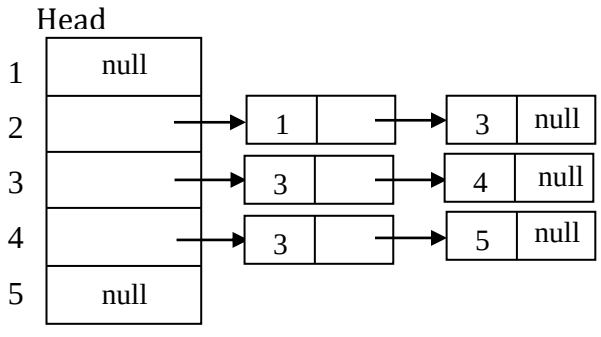
Head

1	3	3	4	3	5
1	2	3	4	5	6

Succ

b. Linked lists

The same principle can be applied using linked lists as follows:



Here too, an additional space should be added for each block in the case of a weighted graph to store the weight of each arc.

4. Graph exploring

4.1 Building predecessors lists

a. Using adjacency matrix representation

Let M the adjacency matrix of a graph of order n. The successors of a given node i can be obtained easily by traversing the row i of the matrix M in $O(n)$ operations, while predecessors are obtained by traversing the column i also in $O(n)$.

b. Using adjacency lists representation

For low density graphs, it is advantageous to encode the graph by linear structures representing the adjacency lists. In fact, the structure requires only $O(m+n)$ memory space and the list of successors requires only $O(d^+(i))$ operations. However, adjacency lists cannot be manipulated efficiently to retrieve the list of predecessors of a node i. To find the list of predecessors, it is necessary to scan the whole graph to detect the nodes of which i is the successor and this requires $O(m)$ operations and $O(nm)$ operations to build predecessor lists of all the nodes. In the case where the list of predecessors is required frequently, it is more advantageous, in terms of time complexity, to construct the inverse graph G^- of G where the lists of successors are simply the lists of predecessors in G. In this case, we need $2(m+n)$ memory space for the two graphs, but this is clearly less than n^2 in the case of low density graphs.

4.2. Graph exploring

The exploration consists in determining the set of descendants of a node s, i.e. the set of nodes located on paths starting from s. The principle of the exploration of a graph can be described as follows: At the beginning we mark the node s, then each time we encounter an arc (x, y) with x marked and y unmarked, we mark the node y.

a. Breadth-First Search (BFS)

In this type of exploration, we implement the set of nodes marked as a queue Q. The nodes are marked in an increasing order: we start with the successors of s, then successors of successors of s, etc. A node x at the head of the queue Q remains until its successors are examined. Meanwhile, any unmarked successor of x is marked and placed at the end of Q.

Algorithm:

```
mark s; enqueue(Q,s)
Repeat  ……N
  x = dequeue(Q)
  for each unmarked successor y of x ……∑d+(i) = M
   enqueue(Q,y)
   mark y
  endFor
until isEmpty(Q)
```

Complexity: O(N+M) ≈ O(M)

b. Depth-First Search (DFS)

In this exploring strategy, the set of nodes marked is implemented by a stack S and the exploration progresses by moving as far as possible along a path of which s is the origin before turning back. The stack S contains the explored nodes and allows the procedure to go back.

Algorithm:

```
mark s ; push(S,s)
repeat
  while exists(unmarked successor y of top(S)) do
    mark y
    push(S,y)
  endWhile
  x = pop(S)
until isEmpty(S)
```

Same complexity as BFS.

c. Example

Let G a directed graph, whose nodes are (s1, s2, s3, s4, s5, s6) represented by the following adjacency matrix. Illustrate BFS and DFS exploring strategies on the graph G, starting by node s1.

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

BFS:

Queue	s1	s6 s3 s2	s5 s4 s6 s3	s5 s4 s6	s5 s4	s5	Empty Queue
Marked nodes	s1	s2,s3,s6	s4,s5				

Dequeue order: s1,s2,s3,s6,s4,s5

DFS:

Stack	s1	s2 s1	s4 s2 s1	s3 s4 s2 s1	s5 s3 s4 s2 s1	s6 s5 s3 s4 s2 s1	s5 s3 s4 s2 s1	s3 s4 s2 s1	s4 s2 s1	s2 s1	s1	Empty Stack
Marked nodes	s1	s2	s4	s3	s5	s6						

Pop Order: s6,s5,s3,s4,s2,s1

5. Shortest path problem

5.1. Typology, algorithms and applications of shortest path problems

Let $G = (X, A, W)$ a weighted and directed graph and consider the length of a path as the sum of the weights of its arcs. The main problem consist in finding paths of minimal length. In this context, the following three problems should be distinguished:

1. Let *s*, *t* two nodes, find a shortest path from *s* to *t*.
2. Find shortest paths from *s* to all graph nodes.
3. Find shortest paths between each pair of nodes.

In this section, we will focus on problem of type 2, by calculating for each node x the value of the shortest path from the starting node to *x*; V[*x*].
There is a family of algorithms that compute V[*x*] in a definitive way for each node x. These algorithms are called **label-fixing** algorithms and the most known is Dijkstra's algorithm. Other algorithms refine the value of each node *x* until the last iteration. The following cases should be distinguished:

- Case W constant. The problem is reduced to that of finding the paths containing the smallest number of arcs which can be solved by a BFS exploration.

- Case $W \geq 0$. The problem can be solved by Dijkstra's algorithm which is of the label-fixing type and its complexity is $O(n^2)$ and can improved using a heap structure with a complexity of $O(nlogn)$.

- Case $W \in \mathbb{R}$. There is an algorithm due to Bellman based on dynamic programming. The time complexity is $O(nm)$ and can be improved to $O(m)$ if the graph does not contain circuits.

The applications of shortest path problems are numerous. In the field of transport, for example, we are interested in the optimal paths from a city x to another city y. In network traffic routing, we talk about OSPF (open shortest path first) protocols.

5.2. Dijkstra's Algorithm

It is a label-fixing algorithm and can only be applied for graphs with **positive** weights. The algorithm fixes the label of each node **x** at each iteration by updating an array of Booleans.

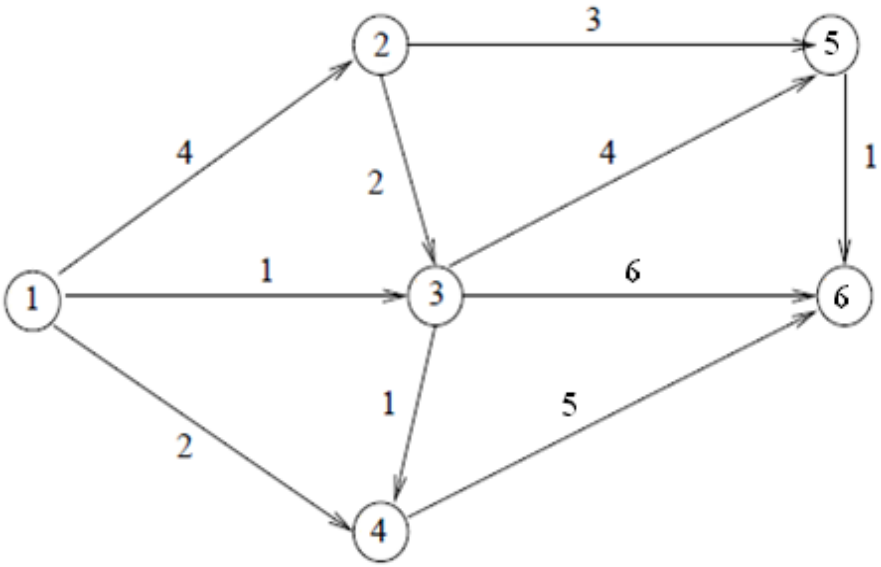
The algorithm uses three arrays; V for values of shortest path of each node, P for storing the shortest path details and D for node fixation. During its initialization phase, the algorithm initializes the arrays V to ∞, P to zeros and D to false. For a given node *s*, we initialize V[*s*] to zero and P[*s*] to s. The main iteration of the algorithm consists of two loops. The first consists in finding an unfixed node of the set V which is minimal. For this, the value ∞ is first assigned to Vmin, then for all *y* ranging from 1 to n for which D[*y*] is false and V[*y*] strictly less than the value Vmin, the algorithm keeps *y* in a variable *x* and assigns V[*y*] to Vmin. If Vmin is not ∞, D[*x*] is replaced by true. The second loop consists in traversing all the nodes *k* successors of x to update the list of the successors. If V[*x*] + W[*k*] is less than V[*y*] then V[*x*] + W[*k*] is assigned to V[*y*]. Finally, *x* is assigned to P[*y*].

Dijkstra's algorithm

```
s: starting node
Initialize array V to +∞          //Shortest path Value
Initialize array P to 0           //Shortest path Detail
V[s]=O
P[s]=s
Repeat
  // Find unfixed node with minimal value in V
  Vmin=+∞
  For i=1 to n do
    if (i unfixed  and  V[i]<Vmin) then  x=i; Vmin=V[i]  endIf
  endFor
  If Vmin<+∞
    Fix x   // Array D
    // Update successors
    For each unfixed successor y of x do
      If V[x]+W[x][y] < V[y]
          V[y] = V[x]+W[x][y];
          P[y] = x
      endIf
    endFor
  endIf
Until Vmin=+∞
```

The complexity of Dijkstra's algorithm is $O(n^2)$ in the worst case. It can be improved to O(nlogn) by using a heap rather than an array for the structure V.

Example: Given the following directed graph, apply Dijkstra's algorithm from node 1, that is to find shortest path from node 1 to all nodes.



Array V (Values)

1	**2**	**3**	**4**	**5**	**6**	**Fixed Nodes**
0	∞	∞	∞	∞	∞	1
	4	1	2	∞	∞	3
	4		2	5	7	4
	4			5	7	2
				5	7	5
					6	6
0	**4**	**1**	**2**	**5**	**6**	**Final Result**

Array P (Paths)

1	**2**	**3**	**4**	**5**	**6**
1	0 1	0 1	0 1	0 3	0 3 5

We deduce shortest paths from node 1:

1→1	0
1→2	4
1→3	1
1→4	2
1→3→5	5
1→3→5→6	6