CPS260/BGT204.1 Algorithms in Computational Biology **11 September 2003**

Lecture 6: Graphs, Representation of Graphs and Traversal Algorithms

Lecturer: Pankaj K. Agarwal *Scribe: Sukhendu Chakraborty*

6.1 Research Projects

A few topics for the research project are discussed below.

- Multiple Sequence Allignment
  - Survey of the known techniques.
  - Development of a new scheme.
  - Implementation of some of the schemes and comparision of their performances.
- Gene regulation or expression
  - Literature review.
  - Clustering techniques.
  - Data mining and classification.
- Protein structures
  - Backbone matching.
- Shape Descriptions
- Secondary structure prediction
  - Statistical techniques.
- Similarity measures for proteins
- Speeding up molecular dynamics *using Monte Carlo simulations* etc.
- Protein - Protein interaction
  - Scorefunctions for docking.
  - Matching shapes of active regions.
  - Classification of binding sites.

6.2 What is a graph?

Graphs are one of the most common mode of representation of relationships between objects. Graphs are used in the analysis of electrical networks; the study of the molecular structure of chemical compounds (perticularly hydrocarbons); the representation of airline routes and communication networks; in planning projects, genetic studies,statistical mechanics,social sciences;and in many other situations. Many 'real world' problems can be formulated and solved easily using graphs.

Before we go any further, let us go through the basic definitions that we encounter quite frequently while discussing graphs.

A graph $G = (V, E)$ consists of a finite set of *vertices* V and a finite set of *edges* E.

- *Undirected graph*: E is a set of unordered pairs of vertices $\{u, v\}$ where $u, v \in V$ i.e. $(u, v) = (v, u)$.

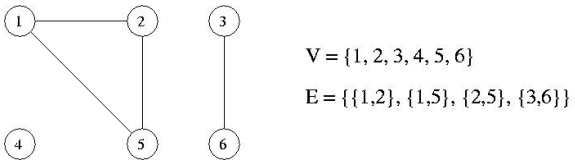

Figure 6.1: An undirected graph.

- *Directed graph* : E is a set of ordered pairs of vertices (u, v) where $u, v \in V$.

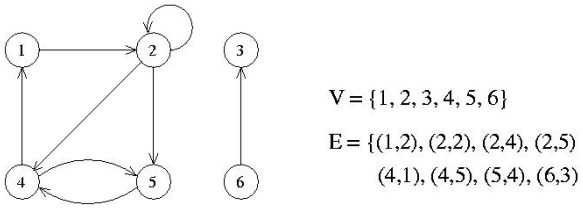

Figure 6.2: A directed graph.

The edge (u, v) is *incident* to u and v. The *degree* of vertex in undirected graph is the number of edges incident to it. The *in (out) degree* of a vertex in directed graph is the number of edges entering (leaving) it. A *path* from u_1 to u_2 is a sequence of vertices $\langle u_1 = v_0, v_1, v_2, \cdots, v_k = u_2 \rangle$ such that $(v_i, v_{i+1}) \in E$ (or $\{v_i, v_{i+1}\} \in E$)

- We say that u_2 is *reachable* from u_1
- The *length* of the path is k
- It is a *cycle* if $v_0 = v_k$

An undirected graph is *connected* if every pair of vertices are connected by a path A directed graph is *strongly connected* if every pair of vertices are reachable from each other.

Graphs appear all over the place in all kinds of applications, e.g:

- Trees ($|E| = |V| - 1$)
- Connectivity/dependencies (house building plans, WWW-page connections, ...)
- Representation of relationships between people,for e.g. *classification tree.*These type of graphs are called *abstract graphs.*

Often the edges (u, v) in a graph have weights $w(u, v)$, e.g.

- Road networks (distances)
- Cable networks (capacity)

Importance of graphs in genetics. Various problems, for eg.*backbone matching,similarity measurements of protein structures etc.* are easily solved by graphs. As an illustration we can consider the following example.
We have a genome which has to be chopped into pieces such that we have *overlapping intervals*of genomes (numbered 1 to 9). See Figure 6.3. This problem can be *transformed* into a graph problem by selecting each interval to be the 'node' of the graph. There exists an edge between two nodes if the two intervals overlap. See Figure 6.4

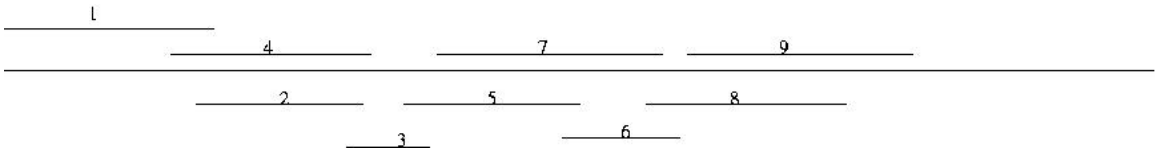

Figure 6.3: A genome to be spliced into intervals

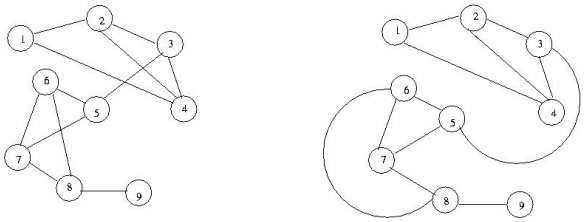

Figure 6.4: The graphical (non-planar and planar) representation of the genome example.

In compact form,we are to prepare a graph $G = (V, E)$ where

- V = set of intervals 1,2,.....9.
- E = { (u,v) such that u and v overlap}

A graph is easy to analyse in 2-dimension and for it to be true,it should be 'drawable' in a plane. Such graphs which can be redrawn in a plane so that the edges do not cross each other are called *planar graphs*. See Figure 6.4.

6.3 Representation of Graphs

For the graphs to be of any use to the 'computational' world, it must be made representable in efficient and elegant form. The following subsections discuss the two most common computational representations of graphs.

Adjacency-matrix representation. We use a $|V| \times |V|$ matrix A where

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

See Figures 6.5 and 6.6.

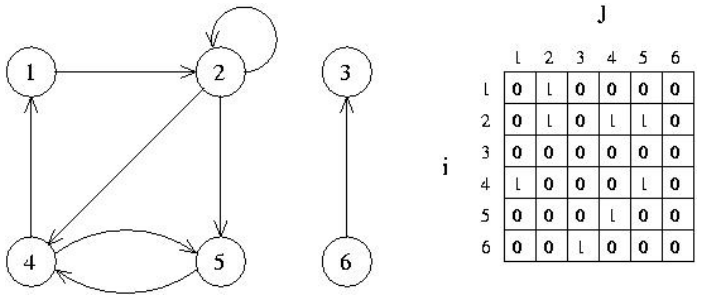

Figure 6.5: Adjacency matrix representation of a directed graph

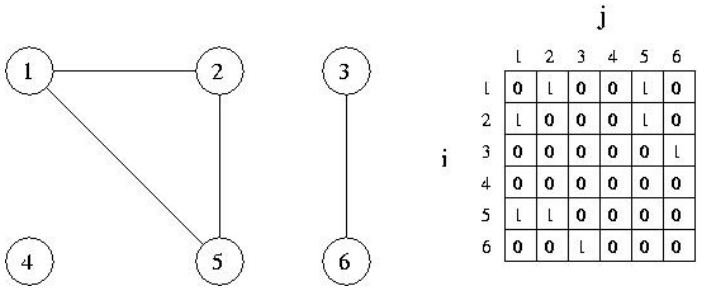

Figure 6.6: Adjacency matrix representation of an undirected graph

For undirected graphs, every edge is stored twice. Hence, space is $O(|V| + 2|E|) = O(|V| + |E|)$. If graph is weighted, a weight is stored with each edge. For undirected graphs, the adjacency matrix is symmetric along the main diagonal (i.e., $A^T = A$). If the graph is weighted, weights are stored in the adjacency matrix instead of 1s.

Adjacency-list representation. We use an array of $|V|$ along with the list of edges incident to each vertex. See Figures 6.7 and 6.8.

For undirected graphs, every edge is stored twice. Hence, space is $O(|V| + 2|E|) = O(|V| + |E|)$. If graph is weighted, a weight is stored with each edge.

Table 6.1 provides a comparison of adjacency-matrix and adjacency-list representations of a graph.

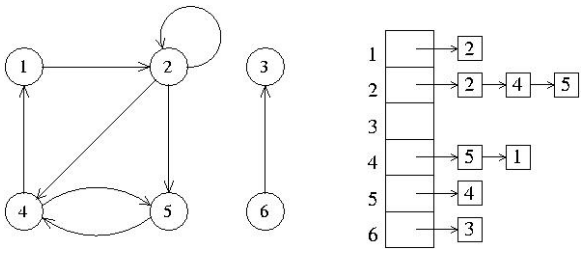

Figure 6.7: Adjacency list representation of a directed graph

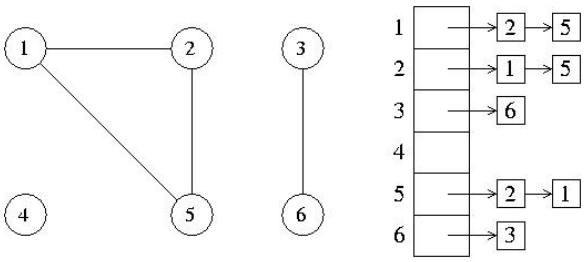

Figure 6.8: Adjacency list representation of an undirected graph

6.4 Graph Traversal Algorithms

- There are two standard (and simple) ways of traversing all vertices/edges in a graph in a systematic way: breadth-first search, and depth-first search.
- We can use them in many fundamental algorithms, e.g., finding cycles, connected components, ...

6.4.1 Breadth-first search (BFS)

BFS corresponds to computing *shortest path* distance (in terms of the number of edges) from s to all other vertices. That is, we start by visiting some source vertex s, then visit all vertices at distance 1, and then visit all vertices at distance 2, and so on.

To control progress of our BFS algorithm, we think about *coloring* each vertex

Adjacency list	Adjacency matrix
$O(\|V\| + \|E\|)$ space	$O(\|V\|^2)$ space
Good if graph *sparse* ($\|E\| << \|V\|^2$)	Good if graph *dense* ($\|E\| \approx \|V\|^2$)
No quick access to (u, v)	$O(1)$ access to (u, v)

Table 6.1: Comparison of graph representations.

- *White* before we start,
- *Gray* after we visit the vertex but before we have visited all its adjacent vertices,
- *Black* after we have visited the vertex and all its adjacent vertices (all adjacent vertices are gray).

We use a FIFO (First In First Out) queue Q to hold all gray vertices—vertices we have seen but are still not done with. We remember from which vertex a given vertex v is colored gray (visit[v]). The algorithm is sketched in Figure 6.9.

```
BFS(s)

  color[s] = gray
  d[s] = 0
  ENQUEUE(Q, s)
  while Q ≠ ∅ do
      DEQUEUE(Q, u)
      for (u, v) ∈ E do
          if color[v] = white then
          color[v] = gray
          d[v] = d[u] + 1
          visit[v] = u
          ENQUEUE(Q, v)
          fi
      color[u] = black
      end-while
```

Figure 6.9: Breadth first search.

The BSF lgorithm runs in $O(|V| + |E|)$ time.

Remarks.

- The edges (visit[v], v), for all $v \in V$ form a tree called the *BFS-tree*.
- $d[v]$ contains length of shortest path (in terms of the number of edges) from s to v.
- We can use the visit array to find the shortest path from s to any given vertex v, by tracing the path backwards from v: v, visit[v], visit[visit[v]], ...

If the graph is not connected we have to start the traversal at all nodes.

```
∀ u ∈ V
        if color[u] = white then BFS(u)
```

We can use algorithm to compute connected components in $O(|V|+|E|)$ time. Figure 6.10 illustrates how BFS works on a graph.

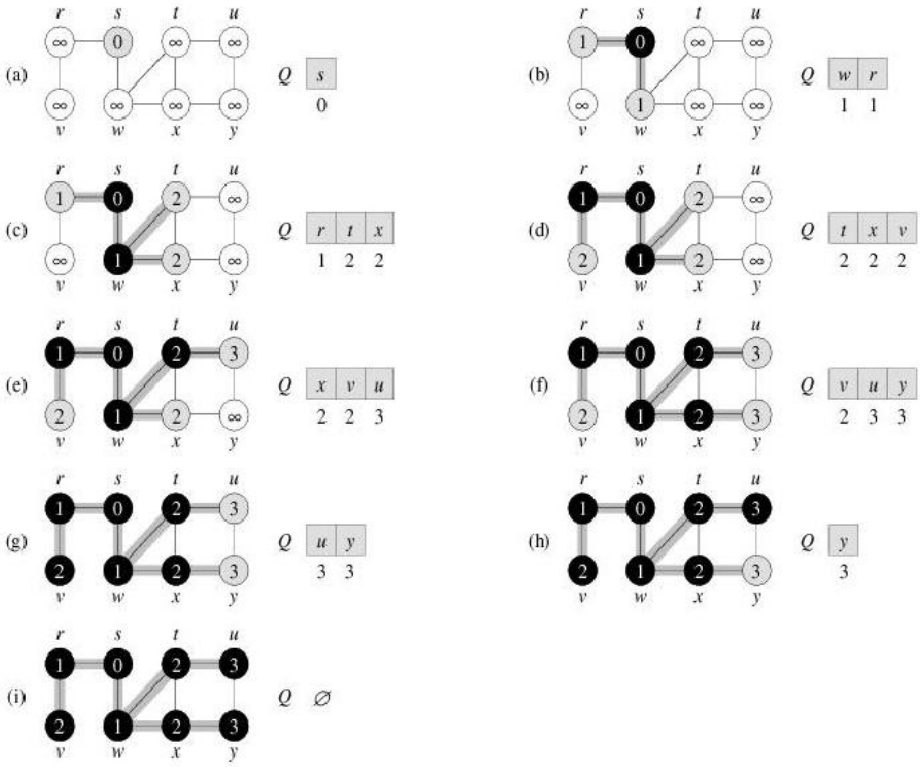

Figure 6.10: Breadth First Search

6.4.2 Depth-first search (DFS)

If we use a stack instead of a FIFO queue Q, we get another traversal order: depth-first search

- We explore "as deeply as possible".
- Backtrack until we find unexplored adjacent vertex.
- Explore as deeply as possible.

  ⋮

Often we are interested in "discovery time" and "finish time" of vertex u

- *Discovery time* ($d[u]$): indicates at what "time" vertex u is first visited.

- *Finish time* ($f[u]$): indicates at what "time" all adjacent vertices of vertex u have been visited.

Instead of using a stack in a DFS algorithms, we can write a recursive procedure. We color a vertex gray when we first meet it and black when we finish processing all adjacent vertices.

```
DFS(u)
  color[u] = gray
  d[u] = time
  time = time + 1
  for (u, v) ∈ E do
      if color[v] = white then
          visit[v] = u
          DFS(v)
          fi
      end-for
  color[u] = black
  f[u] = time
  time = time + 1
```

Figure 6.11: Depth first search.

The DFS algorithm runs in $O(|V| + |E|)$ time. Again, in order to traverse the entire graph, we call the procedure from every vertex.

```
∀ u ∈ V
        if color[u] = white then DFS(u)
```

Remarks. (i) As earlier, the edges $(\text{visit}[v], v)$, for all $v \in V$ form a tree called the *DFS-tree*.

(ii) DFS give valuable information about the structure of a graph e.g. *the paranthesis structure*.

How it works. We first initialize all vertices to white and reset the global counter. We then check each vertex and visit each *white* vertex using DFS. Each call to DFS(u) roots a new tree of depth-first forest at vertex u. A vertex is *gray* if it has been discovered, but not all its edges have been explored! Gray vertices always form a linear chain. A vertex is black after all its edges are explored After DFS is done, every vertex u is assigned: a *discovery* time $d[u]$, and a *finishing* time $f[u]$.

The working of DFS is depicted in Figure 6.12. In the figure,

- each gray vertex is labeled by its discovery time, and
- each black vertex is labeled by both its discovery time and its finish time.

- note: The graphs for which DFS takes maximum space (i.e. stack space for the recursion) are the graphs on which BFS takes minimum space and vice-versa.

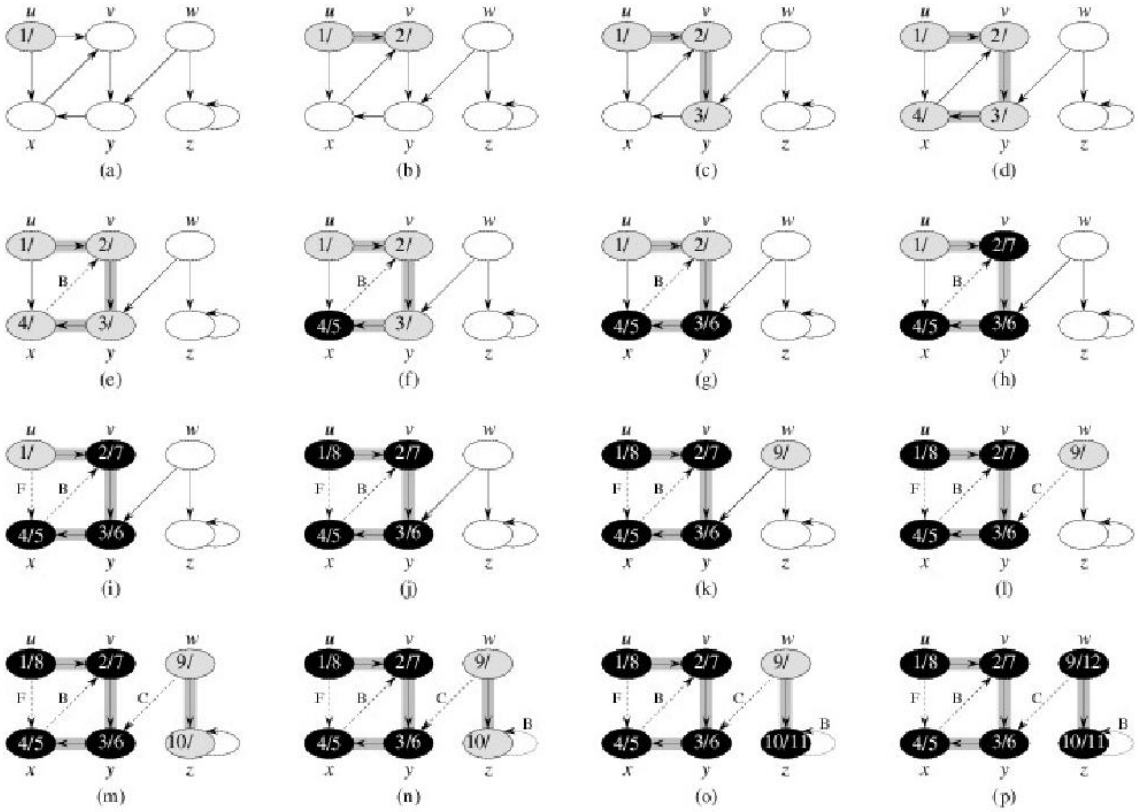

Figure 6.12: Depth first search