CIS 5511. Programming Techniques

Graphs (1)

1. Graphs and their representation

A *graph* consists of vertices (nodes) and edges (links) between them, and represents relations (of the same type) among entities.

Formally, a graph $G = (V, E)$, where V is a set of vertices, and E is a set of edges each being a pair ‹s, d› (from vertex s to vertex d), so $0 \le |E| \le |V|^2$. The "size" of G should be represented by both $|V|$ and $|E|$, though sometime $f(|V|, |E|)$ is simplified to $f(V, E)$.

Under this definition, graph is more general than tree and linear data structures, though it can still be further extended into multigraph, hypergraph, knowledge graph, etc.

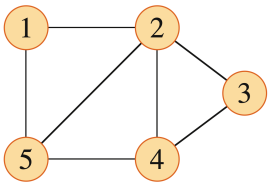
A graph may either be *directed* or *undirected*, depending on whether the relation is symmetric or not.

A *path* is a sequence of end-to-end edges, and its length can be zero or any positive integer. In a directed graph, a *cycle* is a path of non-zero length with the same starting and ending vertex.

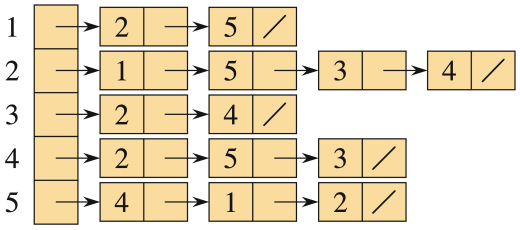
An undirected graph is *connected* if there is a path from every vertex to every other vertex. A directed graph is *strongly connected* if there is a (directed) path from every vertex to every other vertex; it is *weakly connected* if the graph is connected after all directed edges are turned into undirected edges.

The two common ways to represent a graph in a data structure is by an *adjacency matrix* and by an *adjacency list*. The former uses $\Theta(|V|^2)$ space and is better for *dense* graphs, while the latter uses $\Theta(|V| + |E|)$ space and is better for *sparse* graphs. Also, for a directed graph, an adjacency matrix representation allows the edges to be followed in both directions with the same cost, which is not the case for an adjacency list representation.

In the following algorithms, these two representations are unified by using *G.adj[u]* to represent the set $\{v \mid \langle u, v\rangle \text{ is in } G.E\}$.



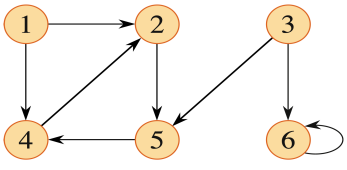
(a)



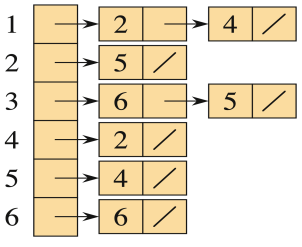
(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)



(a)



(b)

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

(c)

A graph is *weighted* if every edge has a number associated to represent distance, cost, etc., which can be

represented by augmenting the adjacency matrix or list.

2. Search

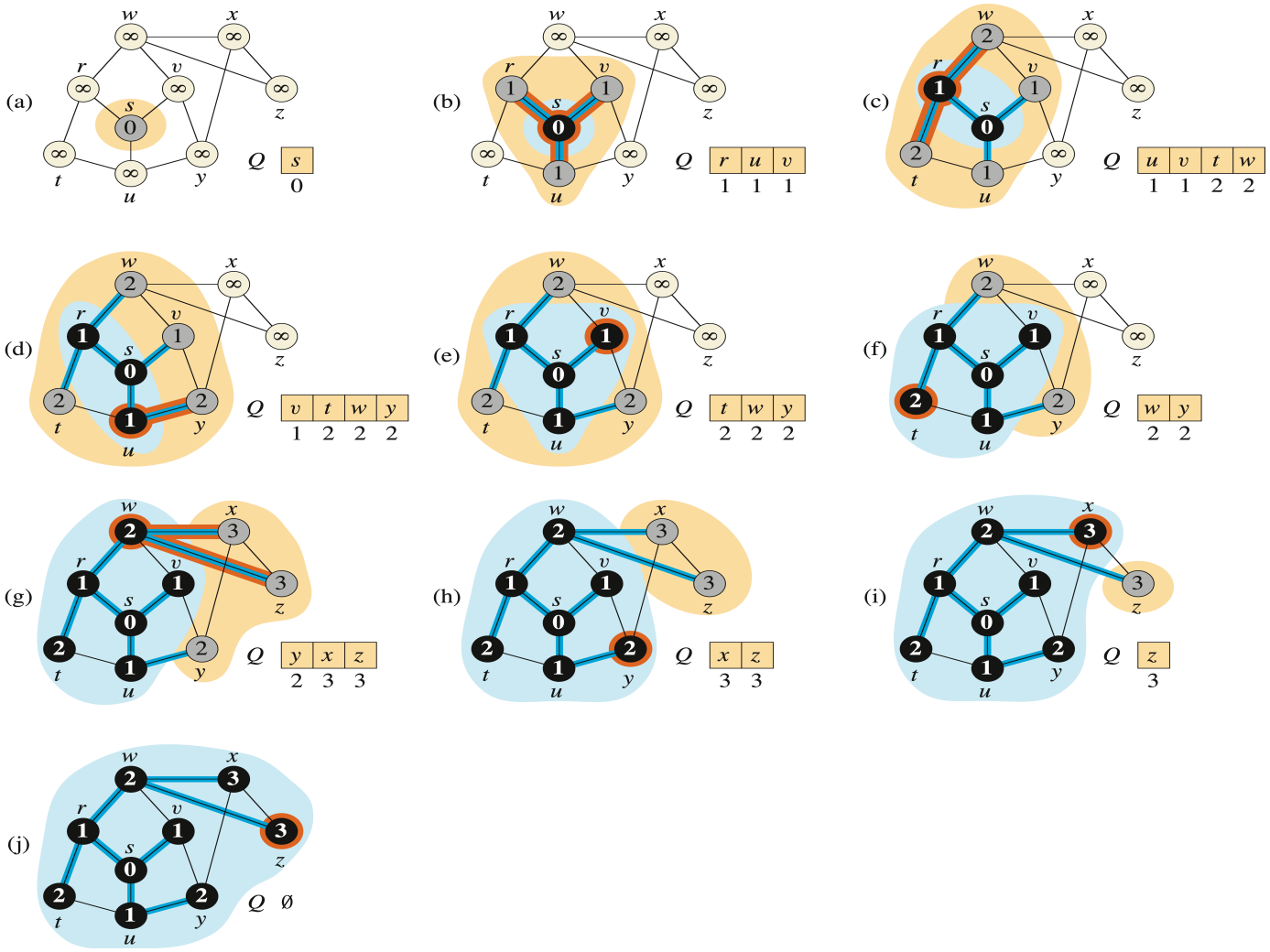
In graphs, various search (traversal) algorithms systematically visit every vertex, normally by following the edges.

Breadth-First Search (BFS): starting from a source vertex, visits its neighbors layer-by-layer with increasing distance. In the following algorithm, a "color" is attached to every node to indicate its status: to be processed (white), being processed (gray), has been processed (black). For each node, its predecessor (π) and distance (d) are recorded (on the path from the source node). A queue Q is used to hold the nodes under processing.

BFS(G, s)

```
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each vertex v in G.Adj[u]   // search the neighbors of u
13         if v.color == WHITE         // is v being discovered now?
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)           // v is now on the frontier
18     u.color = BLACK                 // u is now behind the frontier
```

Example:



Under the assumption that all nodes can be reached from the source node, the running time of BFS is Θ(V + E).

As a by-product, this algorithm also generates a search tree with *s* as root, and finds the shortest path between *s* and every reachable vertex in G. In line 13, if color[v] is not WHITE, then a (undirected) cycle is detected.

If in search an algorithm tries to go as deep as possible in each step, then it is a Depth-First Search (DFS) algorithm.

If the queue in the above BFS algorithm is changed into a stack, the algorithm will do DFS; if the queue is changed into a priority queue, the algorithm will do Best-First Search. In each case, it can only reach the nodes connected to the starting one.

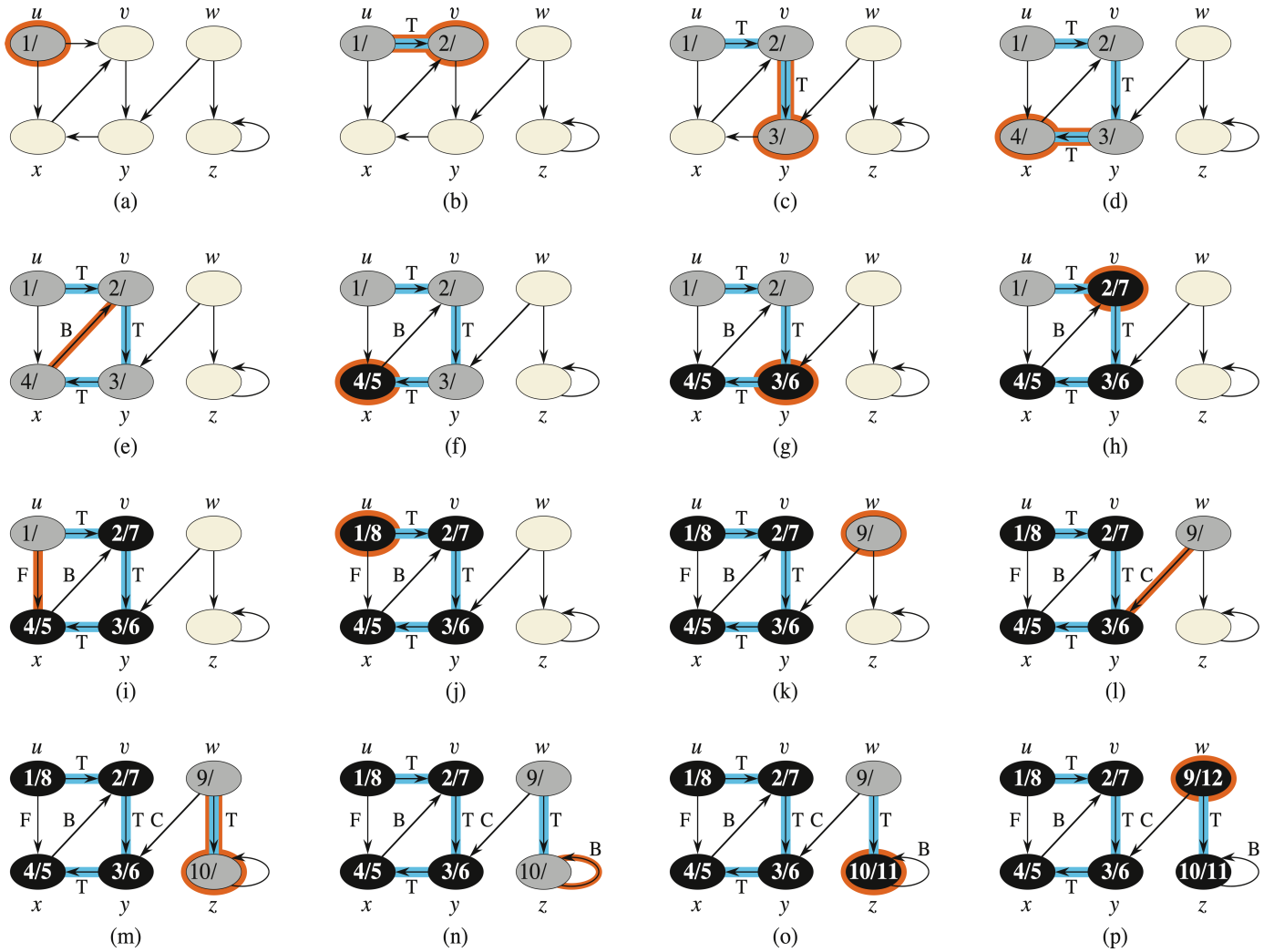
The following is a recursive DFS which does not specify a starting node, and may generate a search forest containing more than one tree when some nodes cannot be reached from the first node. The previous BFS algorithm can be revised to do so, too, though it won't be recursive.

This algorithm uses two time-stamps, *d* and *f*, to record the time of color changing for each vertex. This information will be used in the other algorithms to be introduced later.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1   time = time + 1                  // white vertex u has just been discovered
2   u.d = time
3   u.color = GRAY
4   for each vertex v in G.Adj[u]    // explore each edge (u, v)
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   time = time + 1
9   u.f = time
10  u.color = BLACK                  // blacken u; it is finished
```

Example:



Edges not in the tree are labeled B, F, and C for "back", "forward", and "cross", respectively, depending on whether the two vertices have been linked in the tree by a path.

The running time of DFS is also Θ(V + E).

3. Topological sort

A *topological sort* of a directed acyclic graph ("dag") G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (*u*, *v*), then *u* appears before *v* in the ordering. If the graph is cyclic, no linear ordering is possible.

Topological sorting can be accomplished by repeatedly remove vertex *v* and out-going edges from the graph, under the condition that there is no in-coming edge to *v*, until the vertex set is empty or no longer contains such a vertex (when the graph is cyclic).

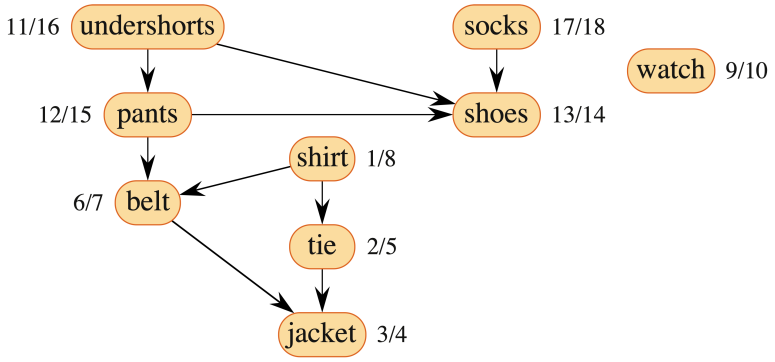
The following solution uses the previous DFS algorithm.

TOPOLOGICAL-SORT(G)
1 call DFS(G) to compute finish times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

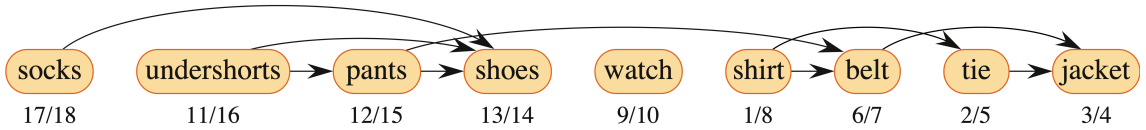
Its complexity is Θ(V + E).

Example:

(a)



(b)



4. Strongly connected components

DFS can also be used to decompose a directed graph into its strongly connected components. In the algorithm, the transpose of G, G^T, is obtained by reversing the direction of all the edges of G. G^T and G have the same strongly connected components, so DFS on both graphs in a proper order will identify them.

STRONGLY-CONNECTED-COMPONENTS(G)
1 call DFS(G) to compute finish times $u.f$ for each vertex u
2 create G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices
in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
separate strongly connected component

Example:

By considering vertices in the second depth-first search (b) in decreasing order of the finishing times that were computed in the first depth-first search (a), the algorithm visits the vertices of the component graph (c) in topologically sorted order.

Since each major step is linear, the algorithm is $\Theta(V + E)$.