13 Graph Search

We can think of graphs as generalizations of trees: they consist of nodes and edges connecting nodes. The main difference is that graphs do not in general represent hierarchical organizations.

Types of graphs. Different applications require different types of graphs. The most basic type is the simple undirected graph that consists of a set \( V \) of vertices and a set \( E \) of edges. Each edge is an unordered pair (a set) of two vertices. We always assume \( V \) is finite, and we write \( |V| \) for the collection of all unordered pairs. Hence \( E \) is a subset of \( (V)^2 \). Note that because \( E \) is a set, each edge can occur only once. Similarly, because each edge is a set (of two vertices), it cannot connect to the same vertex twice. Vertices \( u \) and \( v \) are adjacent if \( \{u, v\} \in E \). In this case \( u \) and \( v \) are called neighbors. Other types of graphs are

- directed: \( E \subseteq V \times V \).
- weighted: has a weighting function \( w : E \rightarrow \mathbb{R} \).
- labeled: has a labeling function \( \ell : V \rightarrow \mathbb{Z} \).
- non-simple: there are loops and multi-edges.

A loop is like an edge, except that it connects to the same vertex twice. A multi-edge consists of two or more edges connecting the same two vertices.

Representation. The two most popular data structures for graphs are direct representations of adjacency. Let \( V = \{0, 1, \ldots, n-1\} \) be the set of vertices. The adjacency matrix is the \( n \times n \) matrix \( A = (a_{ij}) \) with

\[
a_{ij} = \begin{cases} 
1 & \text{if } \{i, j\} \in E, \\
0 & \text{if } \{i, j\} \notin E.
\end{cases}
\]

For undirected graphs, we have \( a_{ij} = a_{ji} \), so \( A \) is symmetric. For weighted graphs, we encode more information than just the existence of an edge and define \( a_{ij} \) as the weight of the edge connecting \( i \) and \( j \). The adjacency matrix of the graph in Figure 50 is

\[
A = \begin{pmatrix} 
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{pmatrix},
\]

which is symmetric. Irrespective of the number of edges, the adjacency matrix has \( n^2 \) elements and thus requires a quadratic amount of space. Often, the number of edges is quite small, maybe not much larger than the number of vertices. In these cases, the adjacency matrix wastes memory, and a better choice is a sparse matrix representation referred to as adjacency lists, which is illustrated in Figure 51. It consists of a linear array \( V \) for the vertices and a list of neighbors for each vertex. For most algorithms, we assume that vertices and edges are stored in structures containing a small number of fields:

```c
struct Vertex { int d, f, pi; Edge *adj};
struct Edge { int v; Edge *next};
```

The \( d, f, \pi \) fields will be used to store auxiliary information used or created by the algorithms.

Depth-first search. Since graphs are generally not ordered, there are many sequences in which the vertices can be visited. In fact, it is not entirely straightforward to make sure that each vertex is visited once and only once. A useful method is depth-first search. It uses a global variable, \( time \), which is incremented and used to leave time-stamps behind to avoid repeated visits.
void VISIT(int i)
1  time++; V[i].d = time;
   forall outgoing edges ij do
2    if V[j].d = 0 then
3       V[j].π = i; VISIT(j)
   endif
   endfor;
4  time++; V[i].f = time.

The test in line 2 checks whether the neighbor \( j \) of \( i \) has already been visited. The assignment in line 3 records that
the vertex is visited from vertex \( i \). A vertex is first stamped
in line 1 with the time at which it is encountered. A vertex
is second stamped in line 4 with the time at which its visit has been completed. To prepare the search, we initialize
the global time variable to 0, label all vertices as not yet visited, and call VISIT for all yet unvisited vertices.

\[
\text{time} = 0; \\
\text{forall vertices } i \text{ do } V[i].d = 0 \text{ endfor; } \\
\text{forall vertices } i \text{ do } \\
\quad \text{if } V[i].d = 0 \text{ then } V[i].\pi = 0; \text{VISIT}(i) \text{ endif}
\]

Let \( n \) be the number of vertices and \( m \) the number of edges
in the graph. Depth-first search visits every vertex once and examines every edge twice, once for each endpoint.
The running time is therefore \( O(n + m) \), which is proportional
to the size of the graph and therefore optimal.

**DFS forest.** Figure 52 illustrates depth-first search by
showing the time-stamps \( d \) and \( f \) and the pointers \( \pi \)
indicating the predecessors in the traversal. We call an edge
\( \{i, j\} \in E \) a tree edge if \( i = V[j].\pi \) or \( j = V[i].\pi \) and a
back edge, otherwise. The tree edges form the DFS forest

![DFS forest](image)

Figure 52: The traversal starts at the vertex with time-stamp 1.
Each node is stamped twice, once when it is first encountered
and another time when its visit is complete.

of the graph. The forest is a tree if the graph is connected
and a collection of two or more trees if it is not connected.
Figure 53 shows the DFS forest of the graph in Figure 52
which, in this case, consists of a single tree. The time-

![DFS forest](image)

Figure 53: Tree edges are solid and back edges are dotted.

stamps \( d \) are consistent with the preorder traversal of the
DFS forest. The time-stamps \( f \) are consistent with the
postorder traversal. The two stamps can be used to decide,
in constant time, whether two nodes in the forest live in
different subtrees or one is a descendent of the other.

**NESTING LEMMA.** Vertex \( j \) is a proper descendent of
vertex \( i \) in the DFS forest iff \( V[i].d < V[j].d \) as well
as \( V[j].f < V[i].f \).

Similarly, if you have a tree and the preorder and postorder
numbers of the nodes, you can determine the relation be-
tween any two nodes in constant time.

**Directed graphs and relations.** As mentioned earlier,
we have a directed graph if all edges are directed. A
directed graph is a way to think and talk about a mathe-
matical relation. A typical problem where relations arise
is scheduling. Some tasks are in a definite order while
others are unrelated. An example is the scheduling of
undergraduate computer science courses, as illustrated in
Figure 54. Abstractly, a relation is a pair \((V, E)\), where

![Course offering](image)

Figure 54: A subgraph of the CPS course offering. The courses
CPS104 and CPS108 are incomparable, CPS104 is a predecessor
of CPS110, and so on.

\[ V = \{0, 1, \ldots, n - 1\} \] is a finite set of elements and
\[ E \subseteq V \times V \] is a finite set of ordered pairs. Instead of
Topological sorting with queue. The problem of constructing a linear extension is called topological sorting. A natural and fast algorithm follows the idea of the proof: find a source, print s, remove s, and repeat. To expedite the first step of finding a source, each vertex maintains its number of predecessors and a queue stores all sources. First, we initialize this information.

for all vertices \( j \) do \( V[j].d = 0 \) endfor;
for all vertices \( i \) do
  for all successors \( j \) of \( i \) do \( V[j].d++ \) endfor;
for all vertices \( j \) do
  if \( V[j].d = 0 \) then ENQUEUE(\( j \)) endif
endfor.

Next, we compute the linear extension by repeated deletion of a source.

while queue is non-empty do
  \( s = \text{DEQUEUE} \);
  for all successors \( j \) of \( s \) do \( V[j].d-- \);
  if \( V[j].d = 0 \) then ENQUEUE(\( j \)) endif
endfor

The running time is linear in the number of vertices and edges, namely \( O(n + m) \). What happens if there is a cycle in the digraph? We illustrate the above algorithm for the directed acyclic graph in Figure 55. The sequence of vertices added to the queue is also the linear extension computed by the algorithm. If the process starts at vertex \( a \) and if the successors of a vertex are ordered by name then we get \( a, f, d, g, c, h, b, e \), which we can check is indeed a linear extension of the relation.

Topological sorting with DFS. Another algorithm that can be used for topological sorting is depth-first search. We output a vertex when its visit has been completed, that is, when all its successors and their successors and so on have already been printed. The linear extension is therefore generated from back to front. Figure 56 shows the same digraph as Figure 55 and labels vertices with time stamps applied by the depth-first search algorithm. The first number gives the time the vertex is encountered, and the second when the visit has been completed.
stamps. Consider the sequence of vertices in the order of decreasing second time stamp:

\[ a(16), f(14), g(13), h(12), d(9), c(8), e(7), b(5) \].

Although this sequence is different from the one computed by the earlier algorithm, it is also a linear extension of the relation.