Effective Software Development Series

Scott Meyers, Consulting Editor

Code *Reading*

The Open Source Perspective

Diomidis Spinellis

Code Reading

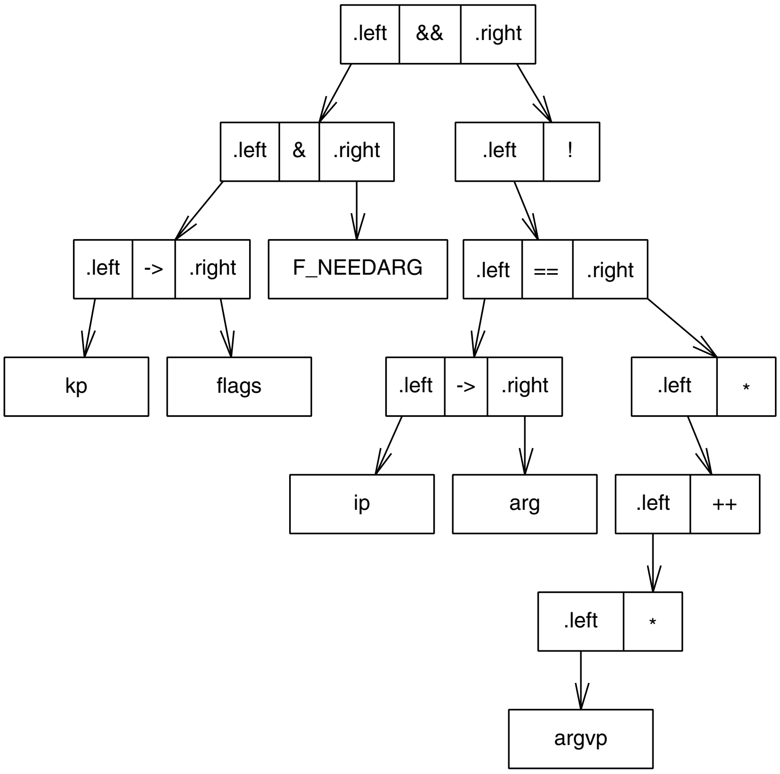

Figure 4.15 Parse tree generated by *lint* for the expression `kp->flags & F_NEEDARG && !(ip->arg = *++*argvp)`

generated by recursively recognizing parts of the language using a *recursive descent parser*[74] or by means of specialized parser generators[75] like *yacc*.

Exercise 4.22 Draw a binary tree that contains the host names in your e-mail address book. Add the names in a random order, not alphabetically. Systematically walk through the tree to derive a sorted listing of the host names. Construct a new tree, adding the host names in the order they appear in the listing you constructed. Comment on the efficiency of searching data in the two trees.

Exercise 4.23 Read about AVL trees in an algorithm textbook and follow the corresponding operations in one of the implementations found in the book's CD-ROM.

Exercise 4.24 Locate in the book's CD-ROM other cases where parse trees are generated. Draw a representative tree for specific input data.

[74]netbsdsrc/bin/expr/expr.c
[75]netbsdsrc/usr.bin/xlint/lint1/cgram.y

4.9 Graphs

A *graph* is defined as a set of *vertices* (or nodes) joined by *edges*. This definition is extremely broad and encompasses data organization structures such as trees (directed graphs with no cycles), sets (graphs with no edges), and linked lists (directed graphs with exactly one edge leading to each vertex). In this section we examine the cases that do not fall in the above categories. Unfortunately, the generality of the graph data structure and the wide variety of requirements of programs that use it conspire to provide us with a bewildering number of options for storing and manipulating graphs. Although it is not possible to discern a small number of "typical" graph data structure patterns, we can analyze any graph data structure by establishing its position on a few design axes. We thus provide answers to the following questions.

- How are nodes stored?
- How are edges represented?
- How are edges stored?
- What are the properties of the graph?
- What separate structures does the "graph" really represent?

We will examine each question in turn.

4.9.1 Node Storage

Algorithms that process a graph need a reliable way to access all nodes. Unlike a linked list or a tree, the nodes of a graph are not necessarily joined together by edges; even when they are, *cycles* within the graph structure may render a systematic traversal that follows the edges difficult to implement. For this reason, an external data structure is often used to store and traverse the graph nodes as a set. The two most common approaches you will encounter involve storing all nodes into an *array* or linking them together as a *linked list*, as is the case in the way nodes are stored in the Unix *tsort* (topological sort) program[76]

```
struc node_str {
    NODE **n_prevp;     /* pointer to previous node's n_next */
    NODE *n_next;       /* next node in graph */
```

[76]netbsdsrc/usr.bin/tsort/tsort.c:88–97

```
    NODE **n_arcs;      /* array of arcs to other nodes */
    [...]
    char n_name[1];     /* name of this node */
};
```

In the case above, nodes are linked together in a doubly linked list using the `n_prevp` and `n_next` fields. In both the linked list and the array representations, the edge structure is superimposed on the storage structure using one of the methods we will outline in the following subsection. In a few cases you will indeed find the graph node set represented and accessed using the edge connections. In such a case a single node is used as a starting point to traverse all graph nodes. Finally, there are cases where a mixture of the two representation methods or another data structure is used. As an example we will examine the graph structure used in the network packet capture library *libpcap*. This library is used by programs like *tcpdump* to examine the packets flowing on a network. The packets to be captured are specified by using a series of blocks, which are our graph's nodes. To optimize the packet capture specification, the block graph is mapped into a tree-like structure; each level of the tree nodes is represented by a linked list of nodes anchored at a level-dependent position of an array. Each block also contains a linked list of edges arriving into it; edges specify the blocks they link.[77]

```
struct edge {
    int id;
    int code;
    uset edom;
    struct block *succ;
    struct block *pred;
    struct edge *next;  /* link list of incoming edges for a node */
};

struct block {
    int id;
    struct slist *stmts;/* side effect stmts */
    [...]
    struct block *link; /* link field used by optimizer */
    uset dom;
```

[77]netbsdsrc/lib/libpcap/gencode.h:96–127

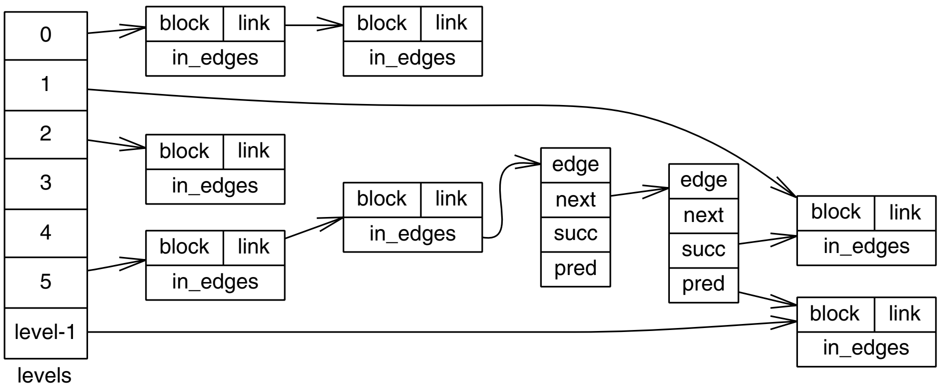

Figure 4.16 Graph nodes accessed through an array of linked lists.

```
    uset closure;
    struct edge *in_edges;
    [...]
};
```

A representative snapshot of the data structure is illustrated in Figure 4.16. In the code example, blocks are linked together in a list using the `link` field, while the edges for each block are anchored at the `in_edges` field; these edges are linked in a list through the `edge next` field, while the blocks they join are specified by the `succ` and `pred` fields. Each list of blocks starts from a different element of a `levels` array. Given this representation, a traversal of all graph nodes is coded as a linked list traversal inside an array loop.[78]

```
struct block **levels;
[...]
    int i;
    struct block *b;
    [...]
    for (i = root->level; i >= 0; --i) {
        for (b = levels[i]; b; b = b->link) {
            SET_INSERT(b->dom, b->id);
```

[78]netbsdsrc/lib/libpcap/optimize.c:150,255–273

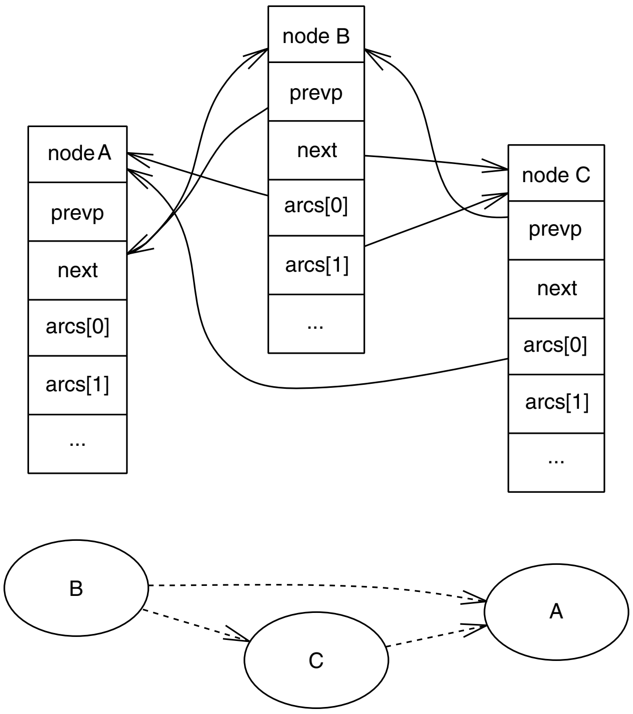

Figure 4.17 A three-node graph and its representation in *tsort*.

4.9.2 Edge Representation

The edges of a graph are typically represented either implicitly through pointers or explicitly as separate structures. In the implicit model, an edge is simply represented as a pointer from one node to another. This is the model used for representing the *tsort* nodes we listed in Section 4.9.1. In each graph node, the array `n_arcs` contains pointers to the other nodes it connects to. You can see a three-node graph and the way its edges are represented as pointers in Figure 4.17.

In many cases, graph edges are represented explicitly so that additional information can be stored in them. As an example, consider the program *gprof*, used to analyze the runtime behavior of programs. (You can find more details about *gprof*'s operation in Section 10.8.) An important part of this information is a program's *call graph*: the way program functions call each other. This is represented in *gprof* by using a graph; the edges (*arcs* in *gprof* terminology) of the graph are used to store information about other related edges.[79]

[79] netbsdsrc/usr.bin/gprof/gprof.h:110–121

```
struct arcstruct {
 struct nl        *arc_parentp;   /* pointer to parent's nl entry */
 struct nl        *arc_childp;    /* pointer to child's nl entry */
 long             arc_count;      /* num calls from parent to child */
 double           arc_time;       /* time inherited along arc */
 double           arc_childtime;  /* childtime inherited along arc */
 struct arcstruct *arc_parentlist;/* parents-of-this-child list */
 struct arcstruct *arc_childlist; /* children-of-this-parent list */
 struct arcstruct *arc_next;      /* list of arcs on cycle */
 unsigned short   arc_cyclecnt;   /* num cycles involved in */
 unsigned short   arc_flags;      /* see below */
};
```

Each arc joins two nodes (represented by a `struct nl`) to store a *call* relationship from a parent to a child, for example, `main` calls `printf`. This relationship is represented by an arc pointing to the respective parent (`arc_parentp`) and child (`arc_childp`) nodes. Given an arc, the program needs to find out other arcs that store similar relationships regarding the parent (caller) or the child (callee) of a node. Links to such arcs are stored in the form of a linked list in `arc_parentlist` and `arc_childlist`. To illustrate this structure we can examine how a small program fragment will be represented. Consider the following code.[80]

```
int
main(int argc, char **argv)
{
[...]
        usage();
[...]
        fprintf(stderr, "%s\n", asctime(tm));
[...]
    exit(EXIT_SUCCESS);
}

void
usage()
{
    (void) fprintf(stderr, "%s%s%s%s",
        "Usage: at [-q x] [-f file] [-m] time\n", [...]
    exit(EXIT_FAILURE);
}
```

[80]netbsdsrc/usr.bin/at