TREES AND GRAPHS: SIMPLE, GENERAL, ABSTRACT, AND EFFICIENT

Boyko Bantchev

Institute of Mathematics and Informatics – Bulgarian Academy of Sciences
boykobb@gmail.com

***Abstract**: The representations of trees and graphs in general, as known from most textbooks on data structures and algorithms or similar sources, are in various ways deficient and outdated. We offer a straightforward approach, based on the notions of set and map, which is at once abstract, general, and efficient, and thus beneficial to the theory, practice, and teaching of programming.*

***Keywords**: graph, tree, representation, algorithm, set, map*

1. Introduction

In mathematics as well as in computing, graphs as combinatorial structures enjoy deserved popularity for their ability to model networks of diverse kind, size, and complexity. In particular, trees, and more precisely, rooted ones, have numerous applications in modelling hierarchies.

The usefulness of graphs in computing is strongly related to how they are represented as data structures, as the feasibility and eficciency — both theoretical and practical — of the algorithms that we perform on graphs for solving various kinds of problems is highly dependent on data representation.

The representations that serve expository purposes, such as in textbooks, are important in two ways. On the one hand, they are expected to ensure clear and unambiguous understanding of how algorithm implementation can be based on them. On the other hand, the very use of representations in teaching tends to them being perceived as a standard to follow and thus they get entrenched in the practice of programming.

Moreover, graph representation in informatics is naturally correlated with the design and implementation of data structures in general — a fundamental part of computing science. In this respect, the theory and practice of representing graphs as data structures is an indication of the current state and trends in the said fundamental field.

However, studying the classic and current textbooks on algorithms and data structures, one can observe that representing graphs (and trees, in particular) in them is insufficient, outdated, or both. There is a significant amount of space for improvement, and in this paper we offer a straightforward, sound and efficient approach, based on simple, few, and general mathematical concepts. But before doing that, we discuss in a little more detail how the current approaches to representing graphs, as seen in textbooks, are imperfect.

2. Representations and shortcomings

Some authors of books on algorithms, such as [1,2,7] choose to present the algorithms in pseudocode or even less formal language, and, accordingly, also do not pay much attention to how data structures, including trees and graphs, need to be represented. In effect, this means that discussion of implementations is avoided, which may be assumed to be motivated by the wish to let the student concentrate on the algorithms' essence. However, such a tradeoff may well be, and often is, unjustified, as the proper understanding of the algorithm may depend substantially on the details of its implementation.

Where representing trees and graphs is indeed discussed, it is often too simple and based on a primitive data structure, such as an array.

For example, a rooted tree is unambiguously defined by mapping each node to its parrent. If, in addition, the nodes are associated with sequential natural numbers, then mapping a node to its parent is simply that of an array index to an array value, where that value is another index. But if a tree is represented like this, navigating from a node to its descendants or siblings, or even only finding the number of descendants, is highly time inefficient, as it requires searching through the array.

Arrays of integers, where integers play the roles of both nodes and links (in the guise of array indices) to nodes are commonly used for representing not only trees but graphs in general. A simple linear array is incapable, or at least very inconvenient for representing even a simple kind of graph, so it is complemented or modified in one way or another. The need to handle nodes with an arbitrary number of neighbours brings to life two-dimensional arrays which must be either rectangular but consuming much unused storage when the graph is sparse, or jagged to fit the number of neighbours for each node separately. In either case one arrives at a structure which lacks flexibility with respect to resizing the graph or changing the connections between its nodes. The very use of integers in different roles can also be confusing in practice.

The above describes scheme of representing graphs is a variant of what is called ‘adjacency list’, as for each node there is a list of nodes adjacent to it. The lists themselves can be linked sequences instead of arrays, but this variant, albeit a little more flexible, is less efficient in terms of both time and storage consumption.

A common method of representing trees is using another kind of tree on which the former one can be modelled. An ordered rooted tree, and even a forest of such trees, is typically represented by a binary tree, where a ‘left’ branch leads to a descendant and a ‘right’ branch links to a sibling. Thus, any representation of a binary tree can serve as a representation of general ordered trees. Furthermore, representing unordered rooted trees is most commonly done by simply using a representation of an ordered rooted tree and not paying attention to the order.

This is the approach taken or suggested in [1,2,6,8]. Although apparently working, it blurs the distinction between the different kinds of trees. Also, due to the linking, which is inherent to binary trees but tends to be excessive for rooted ones, the unifying approach lowers the speed of some algorithms on rooted trees.

Notably, some authors, e.g. [4,7], omit discussing general rooted trees completely, which is, in our opinion, hard to justify.

A generally useful means for designing and presenting data structures, including graphs, is building an abstract representation first. An abstract representation is a collection of datatype declarations, procedures and other data that constitute the programmer's interface to the data structure. It admits further development, as done in [6], into an implementation of the said interface, but when that is omitted, as in [3], then the abstract approach is, just like the use of pseudocode, only a way to evade discussing the actual representation of the data structure.

A notable flaw in most graph representations that one finds in textbooks is that they do not assume any data to be associated with the nodes of the graph. Typically, having once chosen integers (array indexes) to represent nodes, the authors seem to forget that a graph, like any other data structure, is expected to be a container of interlinked data items. Such empty graphs, although sufficient to illustrate the workings of most algorithms on graphs, are nevertheless conceptually and practically inadequate.

Overall, we consider the graph representations widely known from textbooks and elsewhere to be functionally incomplete, too rigid to meet different uses, not always efficient, and also too low-level to conveniently develop and teach algorithms based on them.

3. Programming with sets and maps

The notions of set, map and order are among the most fundamental to mathematics. Finite sets are simple and intuitive, appealing to our very basic mental skills, such as distinguishing between objects, construing a whole out of distinct units, and choosing among many items. Order is what we think of when we consider how things make up a sequence. Maps, being sets of ordered pairs, come next and are almost as easily grasped.

Sets and maps are also not new to programming languages. SETL [5], known since the early 1970s, was the first to introduce ordered and unordered sets, maps, and set-theoretic operations as principal data structures and operations in a programming language. At that time neither computer hardware nor implementation techniques were sufficiently developed to ensure efficient performance for programs in a very high-level language, as SETL was dubbed. But the call for using sets and operations on sets as programming constructs was well heeded and, as time was passing, more and more languages acquired such capabilities.

Today's hardware is hugely more productive, and implementing complex data structures, as well as compiling techniques in general, have advanced to the point that we no more have to consider sets and maps 'very high level' constructs. In fact, they are quickly becoming commonplace. In particular, since its latest editions C++ has begun to offer, through its standard library, excellent support for set-theoretic programming — which is very fortunate, in view of C++ being one of the most widely used and dependable languages of today. Very importantly, the said support comes with guarantees for time efficiency of all set-related operations.

What we just observed encourages rethinking the ways in which we represent complex data structures, so that more use is made of set-theoretic constructs. We consider trees and graphs to be an obvious target of such an effort.

The examples that follow use C++.

4. Representing trees

A (non-empty) rooted tree can be recursively defined as a pair *(N,S)* of a node *N* and a (possibly empty) set *S* of rooted trees. Then *N* is considered a root from which other trees descend, hence are regarded as subtrees. Thus, ultimately, a rooted tree is a collection of hierarchically linked nodes. We usually consider that collection to be finite.

A program representation of a tree would need to associate some data with each node, so we can use the respective datatype as a parameter that marks out a member of the family of tree types that we want to construct. In fact, rather than defining an entire tree, we define a type representing a single node, along with links to descendants. The resulting parameterized datatype is

```
template <typename N>
struct RTnode {N data; unordered_set<RTnode<N> *> * heirs;};
```

where `data` is the value to be stored in a node and `heirs` is a pointer to a set of pointers to the descendants of that node. Note that the datatype definition closely follows the above given informal definition of a tree. We use `unordered_set` for descendants to stress that no order is assumed among them.

A node with no descendants will be represented as an 'empty' pointer, i.e. `nullptr` in C++. As the set of descendants itself can be empty, we do not formally need `heirs` to be a pointer — it could be the set itself. However, then we would have to create and store an empty set for each leaf node, which is less efficient.

The tree node datatype can be split into a definition of a forest and one of a node itself:

```
        template <typename N> struct RTnode;
```

```
template <typename N>
using Forest = unordered_set<RTnode<N> *>;
template <typename N>
struct RTnode {N data; Forest<N> * heirs;};
```

Defining the set of descendants of a node to be a forest where the latter constitutes a separate type — a collection of (pointers to) tree nodes — has the advantage of making it possible to use forests for other purposes as well. In fact, tree and forest are notions dual to each other, each of them being used to define the other, and this version of their program definition reflects the said duality.

Representing trees as above is simple, yet versatile enough to handle trees with any kind of values in their nodes: primitive, such as `int` or `char`, or composite, or pointers to actual values stored elsewhere.

As an example of using the tree definition, here is a procedure performing a preorder traversal of a rooted tree or its subtree. The result is a sequence (`vector`) of the values kept in the visited nodes.

```
template <typename N>
void rt_preorder(RTnode<N> * & n, vector<N> & ns) {
  ns.push_back(n->data);
  if (n->heirs != nullptr)
    for (auto p : *n->heirs) rt_preorder(p,ns);
}
```

Note that if ordered trees must be represented, `unordered_set` in the `Forest`'s definition can be replaced with `set` or `vector` without otherwise changing the program.

5. Representing graphs

Graphs admit various kinds of representations, of which most often used is the previously mentioned ‘adjacency list’. There is no need for the neighbours of a node to actually form a linear list, however — they can make up a set instead.

Let the nodes of a graph store values of type `N`. We introduce the name `Nodes` to denote a set of nodes and define the generic type

```
template<typename N> using Nodes = unordered_set<N>;
```

We can use `Nodes` to represent the collection of neighbours of a graph's node, as well as for other purposes. For example, the type that designates a set of nodes with values of type `char` is `Nodes<char>`.

A graph is defined then as

```
template<typename N> using Graph =
                                     unordered_map<N,Nodes<N>>;
```

— a map from the set of nodes to that of collections of neighbours, and the type of a graph with char nodes is Graph<char>. As with trees, N can be any type.

In order to illustrate the use of the Graph datatype, let us consider implement-tations of breadth-first and depth-first traversal procedures. We assume that the graph to be traversed can be directed or undirected. In the latter case, we must construct our graph so that, if two nodes are adjacent, each of them is in the set of neighbours of the other.

A graph traversal algorithm needs temporary storage to keep track of the already visited nodes. Let that storage be the set vis. Then vis is created with capacity sufficient to eventually hold all the graph's nodes, used in the traversal, and then destroyed. This means that vis must be local with respect to the traversal procedure. In the following implementation the bft_graph procedure creates and destroys vis, while calling another procedure, bft_graph_do, for doing the actual traversal. It also creates a vector nodes for storing the resulting sequence of nodes. nodes and vis are passed to bft_graph_do, along with the graph g and the node x from which the traversal starts.

A breadth-first traversal also typically uses a queue data structure to store nodes that have been visited and are still to be used for reaching from them, using neighbourship, new nodes to visit. That queue, nq, is created, used and destroyed within bft_graph_do. The reason for vis having an outer scope of locality is that if the graph consists of more than one components, we would need to call bft_graph_do once for each of them, keeping the contents of vis between calls: once all the graph's nodes become visited, we know that all the components have been traversed.

```
template<typename N>
vector<N> * bft_graph(Graph<N> & g, N x) {
  auto size = g.size();
  auto nodes = new vector<N>();
  nodes->reserve(size);
  auto vis = new Nodes<N>();
  vis->reserve(size);
  bft_graph_do(g,x,nodes,vis);
  delete vis;
  return nodes;
}

template<typename N>
void bft_graph_do(Graph<N> & g, N x,
                  vector<N> * nodes, Nodes<N> * vis) {
  auto nq = new queue<N>();
  vis->insert(x);
  nq->push(x);
```

```
    while (!nq->empty()) {
      x = nq->front();
      nq->pop();
      nodes->push_back(x);
      for (auto n : g[x])
        if (!contains(vis,n)) {
          vis->insert(n);
          nq->push(n);
        }
    }
    delete nq;
  }
```

The following two procedures, `dft_graph` and `dft_graph_do`, implement depth-first traversal and communicate with each other similarly to the above two. Depth-first traversal here is in its iterative form, which is a bit more challenging to implement than the recursive one. A stack named `prev` is locally defined within `dft_graph_do` and plays a similar role to that of `nq` in `bft_graph_do`: keeping track of visited and not yet to be left nodes. Each item of `prev` is a pair of a node and an iterator, pointing at a member of the set of neighbours of the node currently being visited (`INodes` is defined as `Nodes<N>::iterator`). The node and the iterator together identify at each step which is the next node to visit, if it has not been already visited.

```
  template<typename N>
  vector<N> * dft_graph(Graph<N> & g, N x) {
    auto size = g.size();
    auto nodes = new vector<N>();
    nodes->reserve(size);
    auto vis = new Nodes<N>();
    vis->reserve(size);
    dft_graph_do(g,x,nodes,vis);
    delete vis;
    return nodes;
  }


  template<typename N>
  void dft_graph_do(Graph<N> & g, N x,
                    vector<N> * nodes, Nodes<N> * vis) {
    auto prev = new stack<pair<N,INodes<N>>>();
    for (;;) {
      vis->insert(x);
      nodes->push_back(x);
      for (auto n = g[x].begin();; ++n)
        if (n == g[x].end()) {
          if (prev->empty()) {
            delete prev;
            return;
          }
          tie(x,n) = prev->top();
          prev->pop();
        } else if (!contains(vis,*n)) {
          prev->emplace(x,n);
          x = *n;
          break;
        }
    }
  }
```

It is important to point out that the basic operations on sets, such as insertion and checking for inclusion, are guaranteed to take only a constant time in average, which in turn ensures that traversals have optimal (linear) time complexity with respect to the number of nodes. Also constant-time is removal (not used here).

6. A more elaborate example: Jarník-Prim-Dijkstra algorithm

So far we have discussed graphs that have no data associated with their edges, and the edges themselves were not explicitly represented. Another representation is needed if the edges carry weights or costs.

Let `Weight` be the respective numeric type. We define the type `Nodes` as above and add these types:

```
template<typename N> struct Arc {N node; Weight weight;};
template<typename N> using Arcs = unordered_set<Arc<N> *>;
template<typename N> using EGraph = unordered_map<N,Arcs<N>>;
```

A weighted undirected graph is represented by the type `EGraph`, which is a map from nodes to sets of pointers to `Arc`s, where `Arc` is a half-edge, consisting of node and weight data. Each `Arc` pointed at within a set contains the node and the weight of the edge, connecting that node to the node, associated with the set.

The Jarník-Prim-Dijkstra algorithm finds a spanning tree with a minimal total cost of edges for a given weighted undirected graph by adding edges one by one to an initially empty tree. The edge chosen each time for addition is the one of minimal weight which links a node in the tree to a node not in the tree.

Since the resulting tree is also weighted, we need a new tree node datatype to represent it:

```
   template <typename N>
   struct RTnode {N data;
   unordered_set<pair<RTnode<N> *,Weight> *> * heirs;};
```

The algorithm, although simple in essence, requires some sophistication in order to be implemented efficiently. For lack of space, we do not quote the actual code here. Instead, we only bring attention to the most substantial details.

First of all, we maintain a set of nodes

```
Nodes<N> vis;
```

that have been already added to the spanning tree: this is needed so that we can tell whether an edge can be considered for addition to the tree, i.e. whether exactly one of its ends belongs to the tree.

We also need to know for each member of `vis` precisely where it resides in the tree, so we maintain a map

```
unordered_map<N,RTnode<N>*> nptr;
```

from graph nodes to pointers at tree nodes. As soon as a node is added to the tree, a corresponding member is also added to `nptr`. When a minimal-weight edge and a respective node are being added to the tree, we know through `nptr` to which node of the tree to link them.

Each node not in the tree, which is connected through an edge to a node already in the tree, is mapped to a pair, one member of which is an `Arc` consisting of the respective node in the tree and the weight of the edge between the two nodes, and the other member is an integer, an index in a heap. The map is defined

```
unordered_map<N,pair<Arc<N>,unsigned>> nfree;
```

When a new edge and node are added to the tree, the edges from that node that lead to nodes not in the tree are traversed. If a newly visited node is found in this process, it is included, along with the tree node and the weight of the respective edge, to `nfree`. If a node is not newly visited, then it is already in `nfree`, and then the `Arc` to which it is mapped may be replaced by the currently visited edge, if the latter has smaller weight. Thus, for each node in `nfree`, the weight of the `Arc` to which it is mapped is always the minimal possible. As the edge to be added to the spanning tree at any stage is chosen among those in `nfree`, it is indeed the minimal possible, as needed.

In order to find an edge with a minimal weight efficiently, a priority queue, i.e. a heap, is maintained out of the weights of the `Arcs` in `nfree`. The `unsigned` in `nfree` is the index of the respective weight in the heap. We need to keep the index that way so that we can know, upon changing the weight of an `Arc` to a smaller one, which item of the heap changes, and where that item goes after the heap is repaired. The C++ library implementation of heaps does not allow for such a close inspection of the inner workings of a heap, so we provide a custom implementation.

As with the other algorithms on graphs, the use of sets and maps ensures optimal time efficiency due to the constant in average complexity of the basic operations inclusion, search, and removal for these structures.

Conclusion

We have seen that representing trees and graphs and algorithms on them in programming using fundamental mathematical constructs such as sets and maps is not only feasible but very practical in a modern programming language. Such representations are simple, intuitive, and general, usually closely following the way we describe these structures and algorithms in common mathematical and less formal language. They are also inherently abstract, being at the same time remarkably efficient.

The blend of such virtues makes set-theoretic representations very beneficial to the teaching, practice, and further development of the theory of programming.

In some cases, it opens possibilities for exploring new, closer interactions between different data structures.

References

Th.H. Cormen et al., Introduction to algorithms, 3rd ed., MIT Press, 2009.

S. Dasgupta et al., Algorithms, McGraw-Hill, 2006.

M.T. Goodrich et al., Data structures and algorithms in C++, 2nd ed., John Wiley & Sons, 2011.

G.T. Heineman et al., Algorithms in a nutshell, 2nd ed., O'Reilly, 2015.

J.T. Schwartz et al., Programming with sets: an introduction to SETL. Springer-Verlag, 1986.

R. Sedgewick, Algorithms in C++, 3rd ed., Addison-Wesley, 2001.

R. Stephens, Essential algorithms, Wiley, 2013.

M.A. Weiss, Data structures and algorithm analysis in C++, 4th ed., Pearson, 2013.

ДЪРВЕТА И ГРАФИ – ПРОСТО, ОБЩО, АБСТРАКТНО И ЕФЕКТИВНО

Бойко Банчев

Резюме: *Представянията на дървета и графи от общ вид, познати от повечето учебници по структури от данни и алгоритми и други подобни източници, са в различни отношения непълноценни и остарели. Предлагаме непосредствен подход за представяне, основан на понятията множество и съответствие, който е едновременно абстрактен, общ и ефективен и с това полезен за теорията, практиката и обучението по програмиране.*