Graphs and Trees - And (Non-)Recursive Algorithms, D&C, and Sorting (Ch. 15 into 16)

From page 479 onward, the text can be read apart from the particular 2D grid example that was begun in earlier chapters.

Representing Graphs

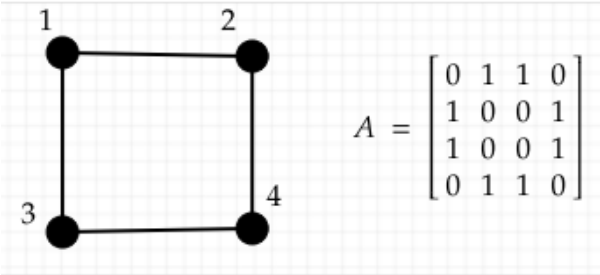
Abstractly, a **graph** is an object $G = (V, E)$ where V is a set of **vertices**, also called **nodes**, and E is a subset of $V \times V$ that consists of pairs (u, v) called **edges**. If the graph satisfies

$$(u, v) \in E \iff (v, u) \in E$$

for all nodes u and v, then it is **undirected**, else it is (properly) **directed**. An undirected graph "Is-A" directed graph in which every edge (u, v) is accompanied by its reversal (v, u), but these classes are most often thought of as separate concepts. Some standard notational conventions that go with graphs:

- Vertices are numbered $i = 1$ to n, where $n = |V|$.
- Edges are numbered 1 up to $m = |E|$. In a directed graph without self-loops, m can go as high as $n(n-1) \sim n^2$. In an undirected graph, the max # of edges is half that, i.e.
$$\binom{n}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2).$$
- The size of the graph is called n or m according to context. Sometimes we use N to mean the "true data size."

Often a graph is represented by another structure, such as a matrix or grid. Here is the 4-cycle graph and its **adjacency matrix**:



A **rectangular grid graph** has $V \subseteq \{1, \ldots, r\} \times \{1, \ldots, s\}$ *minus* some cells that are "blacked out." Its edges are pairs of cells that are adjacent horizontally or vertically [or diagonally]. The text uses this kind of implicit representation of nodes and edges early on. (Networked computations on various kinds of grids is a major research topic of our department associated with Professor Russ Miller and CSE429.)

When some other data point(s) are associated with each node or edge, it pays to represent nodes and/or edges as explicit objects. There are several conventions for this.

1. Use a separate `Node` class and/or `Edge` class. This is common in older texts. *Problem*: the class has public visibility and global scope by default. Different "`Node`" classes might name-clash. Even if not, the "too many little classes" problem.
2. Nest the `Node` class and/or `Edge` class inside the graph class. The text does this on pages 478-479 (though the graph class is less-abstractly called `DrawGraph`). This is standard, but in Scala there is the technicality that inner classes are subsidiary to the enclosing *object*, not the enclosing *class*.
3. Special kinds of graphs have recusive definitions that allow "flattening" the concepts of node and graph.

We have already seen an example of 3. A linked list is a graph whose nodes form a single **path** (or in the case of a circularly liked list, a **cycle** like the one above). In ISR we have followed the standard representation 2. But the native `List` type has no notion of "node"---just the data element directly and recursion back into `List`. The text shows how this is synthesized back in section 12.6 (which I skipped then---but the point will reappear at the end of section 16.3).

Within representation style 2, the choice is whether to represent edges separately or just use tuples. The text on p479 associates data with edges, so it has a separate `GEdge` class from `GNode`. Each node holds a set **`edges`** of its neighbors as a `Set[GEdge]`. One technicality is that an undirected graph needs to recognize that (v, u) is the same edge as (u, v).

`GEdge` has fields `from` and `to`.

Even though representation 2 is usually considered non-recursive, algorithms based on it are often recursive. This can be done with the basic reachability algorithm. But it is also possible to view it non-recursively as a form of **breadth-first search** (**BFS**).

Graph Algorithm: Reachability and Search

To understand what the text is saying on page 480, it pays to insert the idea of "freshly" reaching into the name of the algorithm. The motivation is to eliminate going-around-in-cycles. Here is the pure logic:

A goal end-node e can be freshly reached from a given node u if:

- $u = e$ OR
- (u is fresh, i.e. not an already-visited node, AND
- there is an edge (u, v) to a neighbor v such that---upon declaring u to have been visited, i.e. done-with---we can freshly reach e from v.)

Now we can read the code, using special Scala features (that aren't necessarily recommended):

```
def canFreshlyReach(u: GNode, visited: mutable.Set[GNode] = new Set[GNode]()) = {
   if (u.equals(endNode)) true
   else if (visited.contains(u)) false
   else {
      visited += u
      u.edges.exists(edge => canFreshlyReach(edge.to, visited))
   }
}
```

[Show animations at https://visualgo.net/en --- under Graph Traversals]

Those animations are better described as showing the classic procedural rather than recursive reachability algorithms: **Breadth-First Search** (**BFS**) and **Depth-First Search** (**DFS**).

The idea of BFS is that nodes u are first visited and then (later) **expanded**. Expanding u causes all of the fresh neighbors of u---those not previously visited---to be visited. The node to expand is always the earliest-visited node that hasn't been expanded yet. *Earliest* means we have a first-in, first-out situation, which means using a ***queue*** to manage the visited list. Since we're not putting weights on the edges, we can use a simpler node representation where the **out-neighbors** of u, meaning those v such that (u, v) is an edge, are kept as a list:

```
class Graph[A] {
   class Node(var item: A, val nbhrs: List[Node]) //mutable data, fixed connections
   ...
}
```

The code---**if placed inside the `Graph[A]` class**---is:

```
def classicBFS(start: Node, goal: Node): Boolean = {
   if (start == goal) { return true }
   //else
   var visited = new Queue[Node]()   //Scala Queue is mutable by default
   var expanded = new scala.collection.mutable.Set[Node]()
   visited.enqueue(start)    //so queue is nonempty
   while(!visited.isEmpty) {
      val u = visited.pop()
      expanded += u
      for (v <- u.nbhrs) {
         if (v == goal) {
```

```
            return true
         } else if (!(expanded.contains(v) || visited.contains(v))) {
            visited.enqueue(v)
         }
      }
   } //exit of while loop means no more fresh nodes to visit/expand
   return false
}
```

The algorithm is ***sound*** because if it returns true, it means there is a way to get from `start` to the goal node by traversing edges, so `goal` is reachable from `start`. It is **comprehensive**, and hence correct, because whenever `goal` is reachable from `start`, in k steps, say, then the BFS process will eventually find such a path. [This actually requires an inductive proof on k.]

If this code were outside `Graph[A]`, then we would have the issue of accessing the nested `Node` class. We could write `Graph[...]#Node` (with whatever concrete type is used in place of `A`), and since we're only receiving nodes from the object via `pop()`, we should not have to type-cast them to the graph object. This presumes that `Node` were made public or package-visible. But it is more proper anyway that the BFS code belong to the class---and Scala's general treatment of the inner `Node` class is hinting that it belongs to the graph object itself.

Runtime of BFS

The runtime analysis is highly instructive. We can do the analysis in terms of the number n of nodes and/or the number m of edges. The first thing to notice is that there aren't any simple "for i = 1 to n" type loops. But we can make these two observations:

- Every iteration of the outer while loop pops a node from the queue and puts it in the expanded set---so it can never get back onto the queue. Thus the outer while look can run for at most n iterations, since there are n nodes total.
- The inner for-loop iterates once for each out-neighbor of u, so it runs the **degree** of u number of times. The average degree $deg(u)$ in a directed graph is just the number of edges divided by n, i.e. $\frac{m}{n}$. (In an undirected graph, where you count each (u, v) just once, you get $\frac{2m}{n}$.)

Thus the total number of iterations of both loops combined is at most $n \cdot \frac{m}{n} = m$. Another way to view this fact is that in BFS, no edge ever gets traversed twice, so the total amount of "traversing" is at most the number of edges. If every individual line of code inside the loops were $O(1)$ time, then you could conclude that BFS runs in $O(m)$ time, which in the worst case of $m = \Theta(n^2)$ edges makes $O(n^2)$ time.

However, the test line `(!(expanded.contains(v) || visited.contains(v)))` is not elementary. Both containers can fill up to $\Theta(n)$ nodes. Now because expanded is a `Set`, the `contains`(.) method runs in $O(\log n)$ time at worst [we will see that a hash-table implementation claims $O(1)$ time, but this comes with some fine-print]. But lookup in a `Queue` doesn't promise better than $O(n)$ time.

There is a simple way we could avoid that time sink. We can expand the `Node` class to include a writable flag field, called "`touched`" say. Any visit would set the flag true, and popping the node from the queue would not unset it. So---assuming that nodes were all constructed with the flag false---the test lines would simply be

```
        } else if (!v.touched) {
           visited.enqueue(v)
           v.touched = true
        }
```

Whether we'd need `start.touched = true` earlier in the code too is a good study question. Then the running time for BFS becomes a clean $O(m)$. But---note that the infrastructure of the algorithm still requires random access of individual elements; it is not streamable.

Depth-First Search

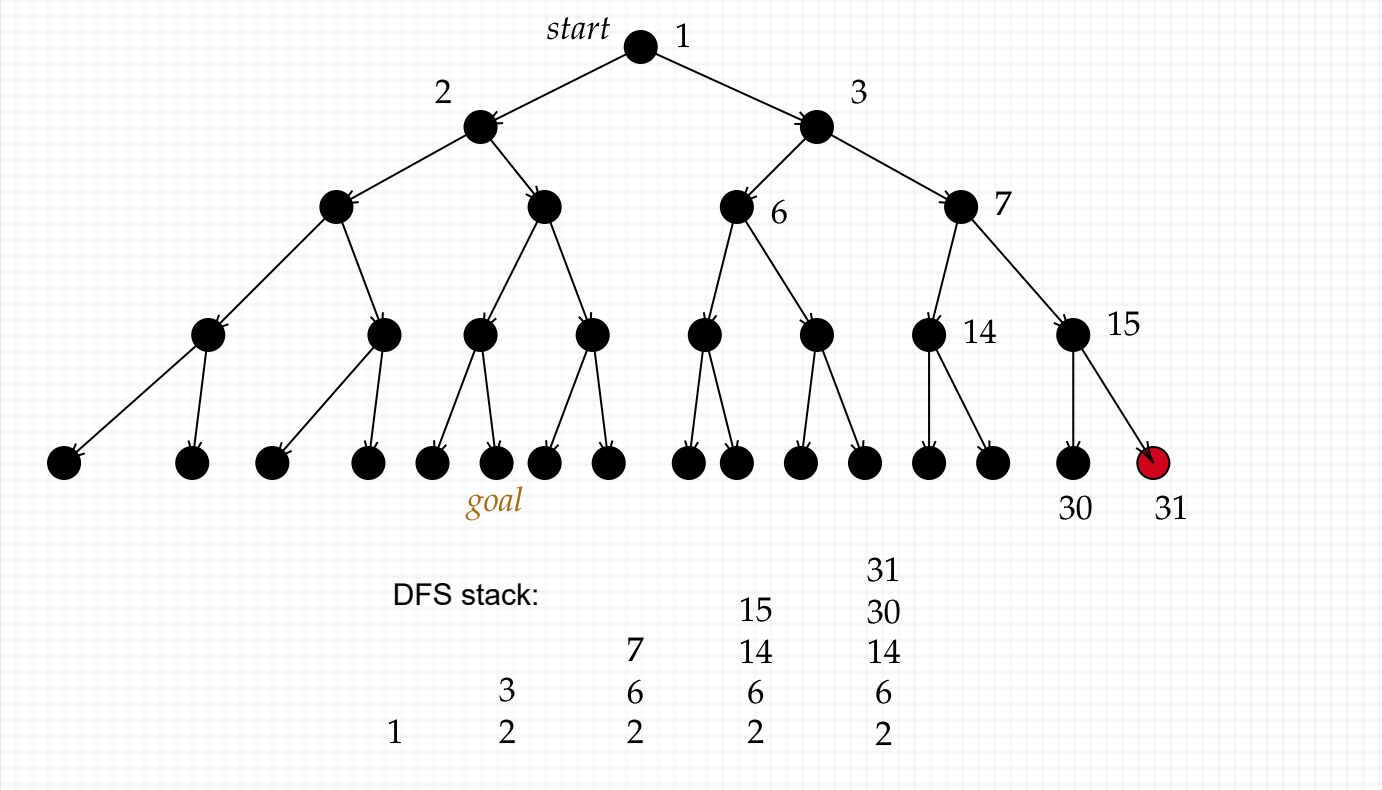
What if we use a Stack instead of a Queue? We can run exactly the same code (with the "touched" update) but just changing the auxiliary data structure:

```
def classicDFS(start: Node, goal: Node): Boolean = {
   if (start == goal) { return true }
   //else
   var visited = new Stack[Node]()   //Scala Stack is mutable by default
   visited.enqueue(start)    //so queue is nonempty
      start.touched = true
   while(!visited.isEmpty) {
      val u = visited.pop()
      for (v <- u.nbhrs) {
         if (v == goal) {
            return true
         } else if (!v.touched) {
            visited.push(v)
            v.touched = true
         }
      }
   } //exit of while loop means no more fresh nodes to visit/expand
```

```
    return false
}
```

The only difference is that now the visited nodes are treated in LIFO order. The running time analysis is entirely similar, because the count of popping from the data structure is the same.

The difference between DFS and BFS is IMHO best appreciated by sketching how each works when started from the root of a binary tree, with the goal node being one of the leaves.



BFS works in a slow-but-uniform way from top to bottom and left to right by row. DFS, however, right away jumps down the rightmost path, "barking down tbe wrong tree" so to speak. After popping the leaves 31 and 30 gives no new out-neighbors, the stack gets down to where it can pop and expand 14, which explores 28 and 29---again with no goal. Eventually the stack gets all the way back down to 2, expanded via the root, and after some more zigs and zags it finds the goal.

Divide And Conquer

The classic Divide and Conquer (**D&C**) situation is when it takes $\Theta(n)$ time to combine the results of two $\frac{n}{2}$-sized halves of the data set. Note that for a tree, $m = n - 1$, so "$O(m)$" and "$O(n)$" are the same thing. The **height** h of a **full** binary tree is $O(\log n)$.

- If every leaf i stores a value x_i, and we just want to compute the maximum value x_T in the tree,

then we can recursively compute the max values x_L in the left subtree of the root and x_R in the right subtree, and combine via $x_M = max(x_L, x_R)$. [If internal nodes store values too, then at the end we get $x_M = max(x_L, x_R, x_{root})$.] The combination time is only $O(1)$ for this final max statement, so the **recursive time equation** is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

which has solution $T(n) = O(n)$. This is just the same as if you just iterate through the n nodes of the tree and take the max on the fly.

- Finding the maximum height of a not-necessarily-full binary tree likewise has $O(1)$ combination time, just to add 1 to the max height of the two subtrees (of the root, or of any non-leaf node recursively speaking), so it gives $O(n)$ time. Once again, this is a case where D&C is used but is not really crucial.
- DFS and BFS can also be done recursively in the two subtrees of the root---then the "combination" is just to tell which subtree had the goal and which did not. So the whole time still comes out linear---the recursion did not add much compared to the classic iterative version.
- But **Mergesort** and **Quicksort**---those are where D&C really shines. They have recursive equation

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

whose solution in general is $T(n) = \Theta(n \log n)$.

The combination step of MergeSort is called `merge(L1,L2)` where L1 and L2 are the two sequences (usually called "lists" but they can be arrays or other containers). They don't have to be exactly the same length. This is a case where the functioning of the algorithm crucially depends on a logical precondition ("**PRE**") of the arguments, namely that L1 and L2 are individually sorted. Here is the code in the text---where L1 and L2 are each an immutable `List`: and `comp` is an `Int` comparison function just like specified for the `ISR` classes (which we will code this for next):

```
def merge1[A](L1: List[A], L2: List[A], comp: (A,A) => Int): List[A]
= (L1,L2) match {
   case (_, Nil) => L1
   case (Nil, _) => L2
   case (x1::rest1, x2::rest2) => if (comp(x1,x2) <= 0) {
      x1::merge1(rest1,L2,comp)
   } else {
      x2::merge1(L1,rest2,comp)
   }
}
```

The text points out on p483 that this version is not **tail recursive**---which is because the outermost operation in the "`if`" and "`else`" cases is `::` rather than the recursive call. The danger it notes is that

with very long lists, the recursion could generate a lot of stack frames and slow execution even if stack-overflow were avoided. It gives a non-recursive version of merge. There is, however, a fairly standard way to make this kind of code tail-recursive using an accumulator parameter (the `comp` part is skipped for simplicity):

```
def merge2[A](L1: List[A], L2: List[A], acc: List[A] = List[A]()): List[A]
= (L1,L2) match {
   case (Nil, Nil) => acc.reverse   //the base case for everything
   case (Nil, x2::rest2) => merge2(Nil, rest2, x2::acc)
   case (x1::rest1, Nil) = > merge2(rest1, Nil, x1::acc)
   case (x1::rest1, x2::rest2) => if (comp(x1,x2) < = 0) {
      merge2(rest1, L2, x1::acc)
   } else {
      merge2(L1, rest2. x2::acc)
   }
}
```

Now the outermost operation on every recursive branch (not counting the base case) is the recursive call to `merge2`, so this is tail-recursive. For reasons similar to the text's version, the target list has to be reversed at the end. Even if the call to reverse needs $\Theta(n)$ units of time, this merely adds to the $\Theta(n)$ time of the merge process itself.

Both versions of `merge` are almost identical to analogous code in the programming language ML. ML compilers by-and-large automatically convert the first version code into the second. The point of tail-recursion is that the compiler is then able to avoid recursion completely, converting everything to an iterative version. Which you could have written to begin with...but the `case-match` based rendition is arguably cleaner.

Instead of what the text has atop page 483, let's see an iterative version of merge using the "first-class iterators" of the `ISR` trait. The second "`B`" generic parameter is needed for a client to tailor the code for the particular containers used.

```
/** REQ: L1 and L2 are sorted in nondescending order and traversed that way.
    NOTE: Will not compile because Iter with () was not included in the ISR trait.
 */
def merge[A, B <: ISR[A] with ClassTag](L1: ISR[A], L2: ISR[A]): B = {
   var ret = new B()     //ClassTag enables this
   var bend = ret.end
   var itr1 = L1.begin
   var itr2 = L2.begin
   while (itr1.hasNext && itr2.hasNext) {
      if (itr1().comp(itr2()) <= 0) {          //ah, this does need () in the trait
```

```
            bend = ret.insert(itr1.next(), bend)  //advances itr1
         } else {
            bend = ret.insert(itr2.next(), bend)  //advances itr2
         }
         bend.next()  //to put "bend" on the end again; maybe ret.end is not O(1) time
      }
      if (!itr1.hasNext) {   //do rest of list 2
         while (itr2.hasNext) { bend = ret.insert(itr2.next(), bend); bend.next() }
      } else {
         while (itr1.hasNext) { bend = ret.insert(itr1.next(), bend); bend.next() }
      }
      return ret
}
```

Regardless of which version of `merge` one uses, the code for `mergeSort` is the same---only needing a method to split the list into two nearly-equal halves. Since merge runs in $\Theta(n)$ time, it is fine for the halving method to take that much time, so the text's "`L.splitAt(L.length/2)`" is AOK. You could also code a similar `halves` method with the iterator counting up to `L.length/2`.

```
def mergeSort[A](L: List[A])(comp: (A,A) => Int): List[A] = {  //or same "B" trick
   if (L.length < = 1) { return L }  //else
   val tup = L.halves
   return merge(mergeSort(tup._1)(comp), mergeSort(tup._2)(comp))
}
```

This is not tail-recursive because `merge` not `mergeSort` is outermost, but that's OK because the **depth** of a D&C recursion by itself is only $O(\log n)$. So there is little danger of stack-overflow here.

One thing about all three versions: The merged list is a ***copy*** of the two given lists. Copies are made of all the data at each depth of the recursion, so $\Theta(n \log n)$ copying is done in all. And even with immutable data, it is real copying: the bottom depths of mergeSort ultimately break the lists into so many little pieces that there is no gain in reusing cells---not like when we were just changing a few cells at the head of a list while making a nominal copy (back in Week 2!). Although it does not affect the theoretical asymptotic $\Theta(n \log n)$ running time (incidentally, mergeSort is never better than that, even on nearly-already-sorted inputs), in practice this copying can be accompanied by a lot of "thrashing." This is so even though---as the text notes on the bottom of pages 483 into 484---you don't need $\Theta(n \log n)$ physical memory to implement it because you can copy back-and-forth between two memory banks at odd and even depth phases. And that brings us to a question not considered by the text:

Is MergeSort stream-friendly?

The answer depends on whether there is a penalty for streams being "paused" for variable time periods. The pause of L1 can be long if a long block of next-lowest elements comes from L2, and vice-versa.

QuickSort