FIT2004: Supplementary notes for lecture 10

Graphs: Dynamic Connectivity

DANIEL ANDERSON [1]

Connectivity in undirected graphs is a simple but widely applicable problem. The problem of determining whether various pairs of nodes are connected can be solved in constant time after some linear time preprocessing of the graph. The problem becomes much more difficult once we allow the graph to modified in between queries. This problem is called the dynamic connectivity problem, and we are going to see an asymptotically optimal solution to the simplest variant of it; incremental connectivity. The incremental connectivity problem can be solved by reducing it to the disjoint set problem, which can be solved in asymptotically optimal time using the Union-Find data structure.

Summary: Shortest paths

In this lecture, we cover:

- Connectivity in undirected graphs
- The dynamic connectivity problem
- The Union-Find disjoint-sets data structure

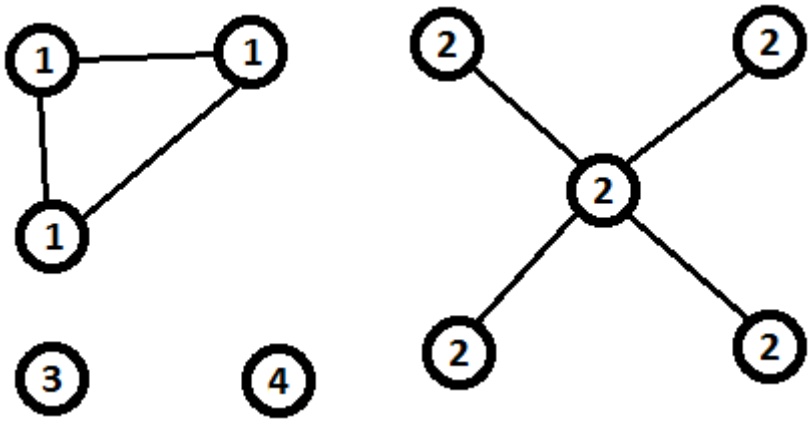
Recommended Resources: Shortest paths

- CLRS, Introduction to Algorithms, Chapter 21
- Weiss, Data Structures and Algorithm Analysis, Chapter 8
- `https://visualgo.net/en/ufds` - Visualisation of the Union-Find data structure

Connectivity in Graphs

Two vertices u, v in a graph G are considered to be *connected* if there exists a path between them. This can be determined in linear time by running a search from u or v and checking whether the other vertex gets visited. Of course, if we wish to check large numbers of pairs of vertices, doing a search each and every time would be redundant and inefficient.

A better solution is to first find the connected components of the graph, which can be done in linear time with a depth-first or breadth-first search.



An undirected graph with each vertex labelled by its connected component number.

[1] FACULTY OF INFORMATION TECHNOLOGY, MONASH UNIVERSITY: `daniel.anderson@monash.edu`. These notes are based on lecture slides by Arun Konagurthu, the textbook by CLRS, some video lectures from MIT OpenCourseware, and many discussions with students.

With the connected components known, each connectivity query can be answered in constant time by simply checking whether the two vertices are assigned to the same component or not.

Algorithm: Connected Components via Depth-First Search

```
function CONNECTED_COMPONENTS(G = (V[1..N], E[1..M]))
    Set component[1..N] = null
    Set num_components = 0
    for u = 1 to N do
        if component[u] == null then
            num_components += 1
            dfs(u, num_components)
        end if
    end for
    return num_components, component[1..N]
end function

function DFS(u, comp)
    visited[u] = True
    component[u] = comp
    for each vertex v adjacent to u do
        if component[v] ==null then
            dfs(v, comp)
        end if
    end for
end function
```

Algorithm: Connectivity Check

```
function CONNECTED(u, v)
    return component[u] == component[v]
end function
```

The problem becomes much more interesting if we want to modify the graph in between queries. One solution is to simply recompute the connected components each time we make a modification, but this will get expensive if we begin to make lots of them. This problem is called the *dynamic connectivity* problem.

The Dynamic Connectivity Problem

The dynamic connectivity problem is the problem of determining whether two vertices u, v in a graph G are connected, while allowing the graph G to be modified, that is, have edges inserted and deleted in between queries. Simply rerunning the connected components algorithm every time we make a modification is inefficient, so we'd like to find a better way to approach the problem.

The fully general dynamic connectivity problem is quite tricky, and is still an active area of research. We will focus our efforts on the simplest but arguably most useful variant of the problem, the *incremental connectivity* problem, in which we are only allowed to add edges, but not remove them.

The incremental connectivity problem

The incremental connectivity problem is the problem of finding a data structure that can support the following two operations:

1. `CONNECTED(u, v)`: Check whether the vertices `u` and `v` are connected.
2. `LINK(u,v)`: Add an edge between the vertices `u` and `v`.

We'd like to perform these operations fast, so simply doing a search per `CONNECTED` query or recomputing every connected component per `LINK` query is too slow.

This problem can be solved using a data structure called a *Union-Find* or *disjoin-set* data structure.

The Union-Find Disjoint-Set Data Structure

The *Union-Find* or *disjoin-set* data structure is a data structure for maintaining a collection of elements, each of which belongs to a single set, and allowing us to merge the contents of some pair of sets together. Each disjoint set is identified by a *representative*, which is some chosen element of the set.

Formally, the operations supported by a Union-Find structure are:

1. `FIND(u)`: Determine which set a particular element is contained in. Usually this means finding the representative of the set.
2. `UNION(u,v)`: Join the contents of the sets containing `u` and `v` into a single set.

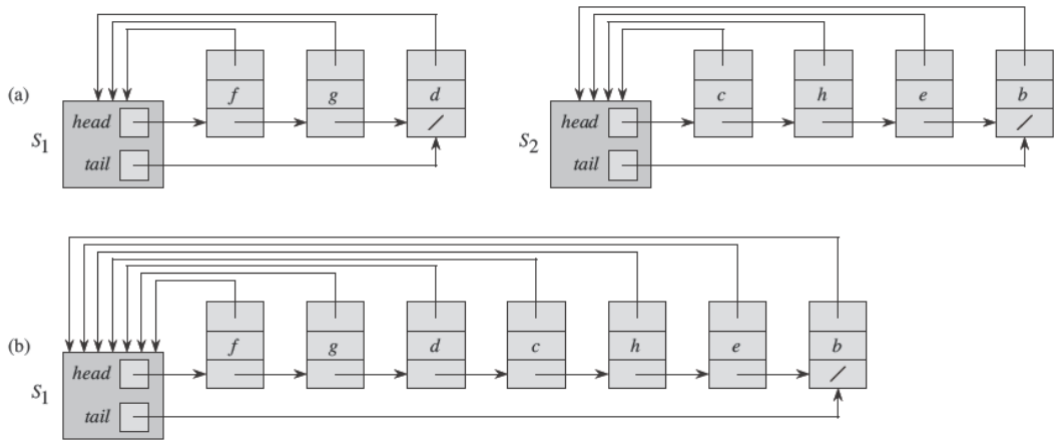
These two operations can be seen as equivalent to maintaining the connected components of an undirected graph, where each item in a set represents a vertex, and each union operation represents the addition of an edge which might connect two components. Checking whether two vertices are in the same connected component then reduces to checking whether they have the same representative, ie. whether `FIND(u) == FIND(v)`.

A naive approach - disjoint set linked lists

One simple but suboptimal way to implement a Union-Find structure is to use linked lists. Each list represents one disjoint set, with each node of the list representing one of the items in the set. The head (first) element of the linked list is chosen to be the representative for each disjoint set.

1. `FIND(u)`: To find the representative of `u`, we simply traverse to the head of the linked list containing `u`.
2. `UNION(u,v)`: To merge the sets containing `u` and `v`, we first check that they are not already contained in the same set (if they are, we do nothing), and if not, simply append the two linked lists containing them together.

Several optimisations can be made to improve the performance of this implementation. For example, we could maintain for each element, a pointer to its representative, making us able to answer `FIND` queries in constant time. This would mean however, that `UNION` now has to do more work in updating each elements representative pointer. We could also choose to track the length of each list, and make `UNION` faster by always choosing to append the smaller list to the larger one. This is called the weighted-union heuristic.



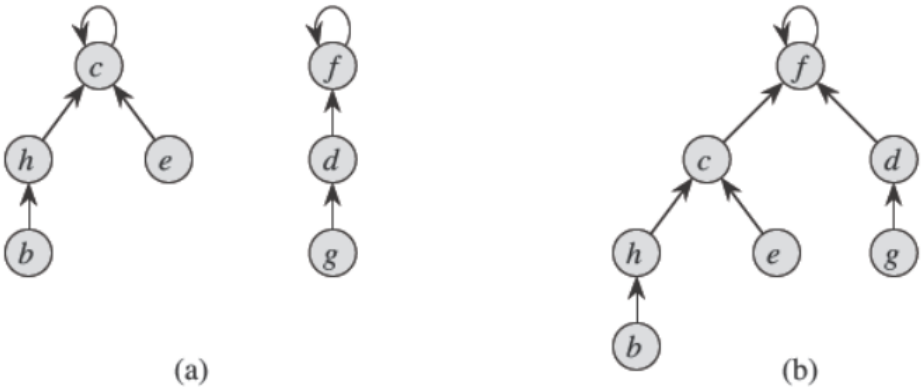
A linked list representation of a disjoint set data structure. The result of merging the two sets shown in (a) is depicted by (b). Figure source: CLRS

It can be shown that using these two heuristics results in a total complexity of $O(N\log(N))$ to perform union $N-1$ union operations in the worst case, which is $O(\log(N))$ on average per union. This is pretty good, but still not optimal. Let's see how we can do even better.

A better approach - disjoint set forests

Rather than represent each set by a linked list, a faster version of Union-Find represents each set by a rooted tree, where each node represents one element and the root node is taken to be the representative element. For each node in the forest, we only need to store a pointer to its parent in the tree, or if the node is a root node, a pointer to itself to indicate it as such.

1. `FIND(u)`: To find the representative of `u`, we simply follow the parent pointers until we reach the root of the tree containing it.
2. `UNION(u,v)`: To merge the sets containing `u` and `v`, we first check that they are not already contained in the same set (if they are, we do nothing), and if not, point the root node of one of the trees to the root node of the other.



A forest representation of a disjoint set data structure. The result of merging the two sets shown in (a) is depicted by (b). Figure source: CLRS

So far, this does not improve on the linked list implementation, since a sequence of union operations may result in a tree of breadth one, which is functionally identical to a linked list. Several optimisations can be made though which result in the forest implementation achieving significantly better performance than the linked list implementation. Indeed, it can even be proven that with these optimisations, the forest implementation of Union-Find is asymptotically optimal, meaning that it is as fast as possible, ie. no data structure can do it faster.

Union-Find using Disjoint-Set Forests (without optimisations)

```
1:  function INITIALISE(N)
2:      Set parent[1..N] = [1..N]
3:  end function
4:
5:  function FIND(x)
6:      if parent[x] = x then
7:          return x
8:      else
9:          return find(parent[x])
10:     end if
11: end function
12:
13: function UNION(x, y)
14:     parent[find(x)] = find(y)
15: end function
```

This unoptimised implementation is still linear time in the worst case due to the potential for trees of breadth one to form from a sequence of `UNION` operations since our implementation chooses to make the tree containing `x` a child of the tree containing `y` arbitrarily. Let's look at some more optimised versions of the data structure.

The path compression technique

The first and most important technique that we can apply to speed up Union-Find is *path compression*. The effect of path compression is very simple, whenever we perform the `FIND` operation to locate the root of a particular node's tree, we might as well point that node directly at the root, so that whatever chain we had to traverse will never have to be traversed again. This has the effect of flattening the tree and removing long chains, which significantly speeds up future `FIND` queries.

The `FIND` Operation using Path Compression

```
1: function FIND(x)
2:     if parent[x] = x then
3:         return x
4:     else
5:         parent[x] = find(parent[x])
6:         return parent[x]
7:     end if
8: end function
```

It can be shown that with path compression, the time complexity of the `FIND` operation is now $O(\log(N))$ on average per query over a sequence of queries.

The union-by-rank technique

Union-by-rank is a similar heuristic to the weighted-union heuristic used in the linked list implementation. In Union-by-rank, we maintain a rank for each tree, which is an upper bound on the height of the tree. When merging two trees together, we choose to always make the tree with a smaller rank a child of the tree with the larger rank. This results in the height of tree growing as least as possible.

The `UNION` Operation using Union-by-Rank

```
1: function INITIALISE(N)
2:     Set parent[1..N] = [1..N]
3:     Set rank[1..N] = 0
4: end function
5:
6: function UNION(x, y)
7:     x = find(x)
8:     y = find(y)
9:     if rank[x] < rank[y] then
10:        parent[x] = y
11:    else
12:        parent[y] = x
13:        if rank[x] = rank[y] then
14:            rank[x] += 1
15:        end if
16:    end if
17: end function
```

It can be shown that with union-by-rank and path compression, that the time complexity of both operations is $O(\alpha(N))$ on average per query over a sequence of worst-case operations, where $\alpha(N)$ is the inverse Ackermann function, an extremely slowly growing function. This is a significant improvement over the $O(\log(N))$ complexity of disjoint set linked lists. This complexity bound also turns out to be provably optimal, in other words, no disjoint-set data structure can possibly do better than this.
