Cosc 201
Algorithms and Data Structures
Lecture 13 (7/4/2025)
Two proofs

Brendan McCane
brendan.mccane@otago.ac.nz
And
Michael Albert

Proving connection and union-find

Connection and union-find

A graph, G, is connected if and only if the union-find instance formed by starting with the vertices of G and taking the union between the endpoints of each edge has only one group.

Proving connection and union-find

Connection and union-find

A graph, G, is connected if and only if the union-find instance formed by starting with the vertices of G and taking the union between the endpoints of each edge has only one group.

Actually we'll prove something stronger.

Connection and union-find

In a graph, G there is a walk from v to w if and only if the union-find instance formed by starting with the vertices of G and taking the union between the endpoints of each edge v and w belong to the same group, i.e., $\mathit{find}(v) = \mathit{find}(w)$.

The proof (I)

- Since we do a union along every edge, the two endpoints of any edge belong to the same group.

The proof (I)

- Since we do a union along every edge, the two endpoints of any edge belong to the same group.
- So, if there is a walk from v to w, say $v = v_0, v_1, \ldots, v_k = w$ then v_0 and v_1 belong to the same group but so do v_1 and v_2 and v_2 and v_3. Eventually, we conclude that v (as v_0) and w (as v_k) belong to the same group.

The proof (I)

- Since we do a union along every edge, the two endpoints of any edge belong to the same group.
- So, if there is a walk from v to w, say $v = v_0, v_1, \ldots, v_k = w$ then v_0 and v_1 belong to the same group but so do v_1 and v_2 and v_2 and v_3. Eventually, we conclude that v (as v_0) and w (as v_k) belong to the same group.
- The remaining problem is to show that if v and w belong to the same group then there must be a walk between them.

The proof (I)

- Since we do a union along every edge, the two endpoints of any edge belong to the same group.
- So, if there is a walk from v to w, say $v = v_0, v_1, \ldots, v_k = w$ then v_0 and v_1 belong to the same group but so do v_1 and v_2 and v_2 and v_3. Eventually, we conclude that v (as v_0) and w (as v_k) belong to the same group.
- The remaining problem is to show that if v and w belong to the same group then there must be a walk between them.
- To do this, we'll show that at any point during the process of doing the union operations and for any vertex v there is always a walk from v to any other vertex u such that $find(v) = find(u)$, i.e., to any other vertex in the same group as v at that point.

The proof (I)

- Since we do a union along every edge, the two endpoints of any edge belong to the same group.
- So, if there is a walk from v to w, say $v = v_0, v_1, \ldots, v_k = w$ then v_0 and v_1 belong to the same group but so do v_1 and v_2 and v_2 and v_3. Eventually, we conclude that v (as v_0) and w (as v_k) belong to the same group.
- The remaining problem is to show that if v and w belong to the same group then there must be a walk between them.
- To do this, we'll show that at any point during the process of doing the union operations and for any vertex v there is always a walk from v to any other vertex u such that $\mathit{find}(v) = \mathit{find}(u)$, i.e., to any other vertex in the same group as v at that point.
- This will be by induction over the sequence of union operations performed.

The proof (I)

- Since we do a union along every edge, the two endpoints of any edge belong to the same group.
- So, if there is a walk from v to w, say $v = v_0, v_1, \ldots, v_k = w$ then v_0 and v_1 belong to the same group but so do v_1 and v_2 and v_2 and v_3. Eventually, we conclude that v (as v_0) and w (as v_k) belong to the same group.
- The remaining problem is to show that if v and w belong to the same group then there must be a walk between them.
- To do this, we'll show that at any point during the process of doing the union operations and for any vertex v there is always a walk from v to any other vertex u such that $\mathit{find}(v) = \mathit{find}(u)$, i.e., to any other vertex in the same group as v at that point.
- This will be by induction over the sequence of union operations performed.
- This is certainly true before the first union operation since $\mathit{find}(v) = \mathit{find}(u)$ means $v = u$ before the first *union* operation.

The proof (II)

- Suppose the property holds and we now do our next union *union*(x, y) for some edge xy of G. Does it still hold?

The proof (II)

- Suppose the property holds and we now do our next union $union(x, y)$ for some edge xy of G. Does it still hold?
- If, previously, $find(x) = find(y)$ then (in terms of groups) nothing has changed, so the property we want still persists.

The proof (II)

- Suppose the property holds and we now do our next union $union(x, y)$ for some edge xy of G. Does it still hold?
- If, previously, $find(x) = find(y)$ then (in terms of groups) nothing has changed, so the property we want still persists.
- Otherwise, the group containing x and that containing y have merged. Suppose that v belonged to one of them (the group containing x say).

The proof (II)

- Suppose the property holds and we now do our next union $union(x, y)$ for some edge xy of G. Does it still hold?
- If, previously, $find(x) = find(y)$ then (in terms of groups) nothing has changed, so the property we want still persists.
- Otherwise, the group containing x and that containing y have merged. Suppose that v belonged to one of them (the group containing x say).
- Can we now walk from v to any vertex in the new group?

The proof (II)

- Suppose the property holds and we now do our next union $union(x, y)$ for some edge xy of G. Does it still hold?
- If, previously, $find(x) = find(y)$ then (in terms of groups) nothing has changed, so the property we want still persists.
- Otherwise, the group containing x and that containing y have merged. Suppose that v belonged to one of them (the group containing x say).
- Can we now walk from v to any vertex in the new group?
- Yes! Because we could already walk to x (and any vertex in its original group), and now we can follow the edge from there to y and then from y to any vertex in *its* original group.

The proof (II)

- Suppose the property holds and we now do our next union $\textit{union}(x, y)$ for some edge xy of G. Does it still hold?
- If, previously, $\textit{find}(x) = \textit{find}(y)$ then (in terms of groups) nothing has changed, so the property we want still persists.
- Otherwise, the group containing x and that containing y have merged. Suppose that v belonged to one of them (the group containing x say).
- Can we now walk from v to any vertex in the new group?
- Yes! Because we could already walk to x (and any vertex in its original group), and now we can follow the edge from there to y and then from y to any vertex in *its* original group.
- So, by induction, the property holds after each and every *union* operation (including the last one, which is when we need it).

Is union-find the correct way to determine if a graph is connected?

- Probably not!

Is union-find the correct way to determine if a graph is connected?

- Probably not!
- Union-find deals with the *dynamic* addition of edges and analyses the evolution of the groups that they form.

Is union-find the correct way to determine if a graph is connected?

- Probably not!
- Union-find deals with the *dynamic* addition of edges and analyses the evolution of the groups that they form.
- If we have a *static* (in the sense of unchanging) graph **we just don't care** what the connectivity status is after we've looked at half the edges. So union-find is probably doing too much work.

Is union-find the correct way to determine if a graph is connected?

- Probably not!
- Union-find deals with the *dynamic* addition of edges and analyses the evolution of the groups that they form.
- If we have a *static* (in the sense of unchanging) graph **we just don't care** what the connectivity status is after we've looked at half the edges. So union-find is probably doing too much work.
- To find the set of vertices, c we can reach from v by walks we just modify the BFT a bit (see next slide).

Is union-find the correct way to determine if a graph is connected?

- Probably not!
- Union-find deals with the *dynamic* addition of edges and analyses the evolution of the groups that they form.
- If we have a *static* (in the sense of unchanging) graph **we just don't care** what the connectivity status is after we've looked at half the edges. So union-find is probably doing too much work.
- To find the set of vertices, c we can reach from v by walks we just modify the BFT a bit (see next slide).
- In common graphics contexts (where we are working in a graph which is a grid) variations on this idea include the flood fill algorithm.

BFT to find the vertices reachable from *v*

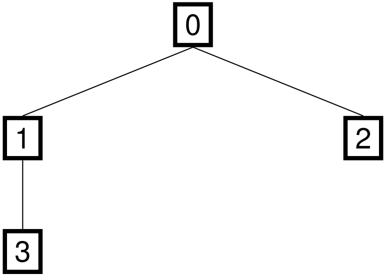
q: an initally empty queue.
c: an initially empty set of vertices.

```
q.addAll(v.neighbours), c.add(v)
while q is not empty do
    w ← q.remove(), c.add(w)
    for n in w.neighours do
        if n is not in c then
            q.add(n), c.add(n)
        end if
    end for
end while
```

Quiz question

Which of the following orderings of the vertices is a valid breadth-first traversal of the given graph starting at node 0?

1. 0, 1, 2, 3
2. 0, 1, 3, 2
3. 0, 3, 2, 1
4. 0, 3, 1, 2



The weighted shortest-path algorithm works?

The data structures associated with the weighted single-source (*base*) shortest-path algorithm are:

weights A map from vertices, *w*, to integers that represents the known weight of the least-weight path from *base* to *w*. Vertices are supposed to be added to this map in increasing order of weight.

parent A map from vertices, *w*, to vertices that represents the parent of *w* along the least-weight path from *base* to *w*.

pq A min-priority queue of edges. The source of each edge in *pq* is a vertex of known weight, and the priority of the edge is the sum of that weight and the weight of the edge.

Our job is to analyse the process of the algorithm and see that at all times these conditions are satisfied.

Initialisation

The intialisations are:

weights	The weight of *base* is set to 0.
parent	The parent of *base* is set to some “end of path” marker
pq	Each edge with source *base* is added.

These are all appropriate for the algorithm (i.e., satisfy the conditions).

Polling the queue

When we poll the queue, we consider the target, t, of the edge removed. There are two cases:

- Its weight is already known, in which case we continue.
- Its weight is not yet known, in which case we assign it.

We need to check that both these are appropriate actions. To do this we proceed by induction on the number of keys in *weight* and add the condition that, if there are k keys in *weight* then these are the k least-weight neighbours of *base*.

Known weight case

The only thing that could go wrong is if we had somehow found a better path to t than the *weight* (and *parent*) maps recognise.

Known weight case

The only thing that could go wrong is if we had somehow found a better path to t than the *weight* (and *parent*) maps recognise.

For this to happen the source, s of that edge has to have lesser weight than the weight of t. So, that edge was added to *pq* before we knew the weight of t. But then, the edge was already in *pq* at the time we determined the weight of t and would have been preferred to the choice we allegedly made.

Known weight case

The only thing that could go wrong is if we had somehow found a better path to t than the *weight* (and *parent*) maps recognise.

For this to happen the source, s of that edge has to have lesser weight than the weight of t. So, that edge was added to *pq* before we knew the weight of t. But then, the edge was already in *pq* at the time we determined the weight of t and would have been preferred to the choice we allegedly made.

So that can't happen and it's safe to just continue.

Unknown weight case

We now need to convince ourselves that t is the next-closest vertex to *base*.

Unknown weight case

We now need to convince ourselves that *t* is the next-closest vertex to *base*.

Whatever that vertex is, the path to it must consist of a path to a vertex of already known weight plus a single edge. But, all these edges are already in *pq*.

Unknown weight case

We now need to convince ourselves that *t* is the next-closest vertex to *base*.

Whatever that vertex is, the path to it must consist of a path to a vertex of already known weight plus a single edge. But, all these edges are already in *pq*.

The edge we've just removed is the one of least priority, i.e., representing the path of least weight – which is exactly what we want!

Unknown weight case

We now need to convince ourselves that *t* is the next-closest vertex to *base*.

Whatever that vertex is, the path to it must consist of a path to a vertex of already known weight plus a single edge. But, all these edges are already in *pq*.

The edge we've just removed is the one of least priority, i.e., representing the path of least weight – which is exactly what we want!

Huzzah!