1 Improving some of the digraph algorithms

Topological Sort: recall that this algorithm on being given a DAG produces a linear ordering of its vertices consistent with the partial ordering defined by its edges. The algorithm given in the last class was:

```matlab
Topological Sort (D)
% Input : DAG D
% Output : list representing a linear order consistent with D
% Effect : destructive algorithm - destroys D
  if D empty
    return NULL
  else
    v = FindSink (D)
    remove v from D
    return v + TopologicalSort(D)
```

FindSink($D'$) costs $\Theta(\min(|V(D')|,|E(D')|))$ in the worst case when applied to digraph $D'$. If we can assume that we are dealing with digraphs satisfying $|E| = \Theta(|V|)$, then Topological Sort (D) then costs $\Theta(|V|^2)$. If instead we assume the digraphs satisfy $|E| = \Theta(|V|^2)$, then Topological Sort(D) costs $\Theta(|V|^3)$ instead.

But what if we take advantage of the recursive nature of the problem?

The algorithm below is non-destructive, though it uses more (though still a linear amount of) space than the one above.

```matlab
Topological Sort (D)
% Input : DAG D
% Output : list L representing a linear order consistent with D
% Uses arrays indexed by vertices of D; OutDegRest(v) has the # edges
% from v to unmarked vertices, MARKED(v) is true iff v is ‘dealt with’.
% Assumes that the digraph offers Theta(1) functions OutDegree and InDegree
% that states which vertices are adjacent to and from a given vertex.
% Vertices are marked as they are added to the output list.
% Telling if a vertex is marked is implemented so it costs Theta(1).

Create empty queue Sinks of sink vertices.
Create empty list L
```
for each vertex \( v \) in \( D \)
\[
\text{OutDegRest} \left( v \right) = \text{OutDegree} \left( v \right)
\]
if \( \text{OutDegRest} \left( v \right) = 0 \)
\[
\text{MARKED} \left( v \right) = \text{TRUE}
\]
\[
\text{Sinks}.\text{enqueue} \left( v \right)
\]
\[
\text{L}.\text{add} \left( v \right)
\]
else
\[
\text{MARKED} \left( v \right) = \text{FALSE}
\]

while \( \text{NOT Sinks.empty} \)
\[
\text{v} = \text{Sinks.dequeue}()
\]
for each vertex \( w \) adjacent to \( v \) (i.e. \( w \rightarrow v \)) such that \( \text{MARKED} \left( w \right) \) is \( \text{FALSE} \)
\[
\text{OutDegRest} \left( w \right) = 0
\]
if \( \text{OutDegRest} \left( w \right) = 0 \)
\[
\text{MARKED} \left( w \right) = \text{TRUE}
\]
\[
\text{Sinks}.\text{enqueue} \left( w \right)
\]
\[
\text{L}.\text{add} \left( w \right)
\]
return \( L \)

This costs \( \Theta(|V| + |E|) \) time. Note that if the structure \( \text{Sinks} \) ever ends up empty while there are still edges left, then those edges must include a cycle.

## 2 Implementing a partial order

Suppose we have some partial ordering relation \( \leq \) on the elements of a set \( S \). In other words, for any \( a, b, c \in S \) we have \( a \leq a \) (reflexivity), \( a \leq b \) AND \( b \leq a \Rightarrow a = b \) (anti-symmetry) and \( a \leq b \) AND \( b \leq c \Rightarrow a \leq c \).

We want a class with the following public member functions:

- **create** \((S)\) : initializes with set \( S \) and no relations.
- **makeLEQ** \((a, b)\) : adds a relation \( a \leq b \).
- **isLEQ** \((a, b)\) : returns \( \text{TRUE} \) iff \( a \leq b \).

This is much easier to implement than it looks, if we already have a digraph data structure with some basic operations.

**create**\((S)\)
\[
\text{create digraph } D \text{ with vertices } S \text{ and no edges}
\]

**makeLEQ**\((a, b)\)
\[
\text{if isLEQ} \left( b, a \right) \text{ AND } a \not= b
\]
\[
\text{ERROR} : \text{BREAKING ANTISYMMETRY}
\]
else
\[
\text{add edge } (a, b) \text{ to } D
\]

**isLEQ**\((a, b)\)
\[
\text{return } \text{TRUE} \text{ iff there is a path from } a \text{ to } b
\]
3 Implementing equivalence relations

Suppose we have some equivalence relation $\sim$ on the elements of a set $S$. In other words, for any $a, b, c \in S$ we have $a \sim a$ (reflexivity), $a \sim b \Rightarrow b \sim a$ (symmetry) and $a \sim b$ AND $b \sim c \Rightarrow a \sim c$. We want a class for which we can define such relations and then ask if two elements are in the same equivalence class.

The data structure should have these public member functions:

- **create** $(S)$ : initializes with set $S$ and no relations.
- **makeEqual**$(a, b)$ : adds a relation $a \sim b$.
- **eqClass** $(a)$ : return the equivalence class of $a$ (what is returned doesn't really matter as long as eqClass(a) == eqClass(b) iff $a$ and $b$ are in the same equivalence class.
- **areEqual**$(a, b)$ : returns TRUE iff $a$ and $b$ are in the same equivalence class.

**Solution 1**: have an array `EquivClass` indexed by vertices.

```plaintext
create(S)
    for each v in S
        EquivClass[v] = v

eqClass(a)
    return EquivClass[a]

makeEqual(a, b)
    ac = eqClass (a)
    bc = eqClass (b)
    for each v in EquivClass
        if EquivClass[v] == ac
            EquivClass[v] = bc

areEqual(a, b)
    return (eqClass(a) == eqClass(b))
```

The cost of find is $\Theta(1)$, makeEqual is $\Theta(|V|)$ if `EquivClass` is implemented using a random access structure such as an array or vector.

**Solution 2**: use an undirected graph with a vertex for each element of $S$ and an edge $ab$ if we are told that $a \sim b$. Two vertices are then equivalent iff they are in the same connected component. To implement `eqClass`, each vertex will also need to have a label, and all vertices in the same connected component will have the same vertex label.

Suppose there is a function `ConnCompt` $(G, v)$ that returns the set of all vertices in the component of graph $G$ containing $v$. Since this can be got by calling DFS Traversal on $G$ starting from $v$, which is linear in the number of vertices $|C|$ in the component $C$, the expected cost of `ConnCompt(G, v)` is $\sum_{\text{components } C \text{ of } G} |C|$, which is $\Theta(|V|)$ in the worst case.

```plaintext
create(S)
    create empty graph G
```
for each v in S
    make v a vertex of G
    label(v) = v

eqClass(a)
    return label(a)

makeEqual(a, b)
    if label(a) != label(b)
        for each v in ConnCompt(G, a)
            label(v) = label(b)
        add edge ab to G

areEqual(a, b)
    return (label(a) == label(b))

The advantage of this is that it ‘remembers’ the original relations specified, which is important for some applications. Solutions 1 and 3 do not do this, which is why they cannot offer a function that removes a relation like this implementation could.

Solution 3: Use the union-find data structure. Suppose that we have an array NEXT indexed by elements of S. This represents the equivalence classes of S as illustrated in the example below. Below right is the data structure the NEXT array below left represents – four directed trees with one sink each. Each tree represents an equivalence class labelled by the sink vertex, so that the partition of S is \{ \{1\}, \{4,8,10,3,2\}, \{5\}, \{6,9,7\} \}.

<table>
<thead>
<tr>
<th>NEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>----</td>
</tr>
</tbody>
</table>

create(S)
    for each s in S
        NEXT(s) = NULL

eqClass(a)
    b = a
    while NEXT(b) not = NULL
        b = NEXT(b)
    return b

makeEqual(a, b)
    NEXT(eqClass(b)) = eqclass(a)

areEqual(a, b)
return (eqClass(a) == eqClass(b))

In more standard notation, the makeEqual operation is called ‘union’ as it defines a new equivalence class as the union of two existing ones, and the eqClass operation called ‘find’ as it finds the representative element of a class containing a given element.

makeEqual and areEqual both have the same asymptotic cost as eqClass, Unfortunately, eqClass costs $\Theta(|S|)$ in the worst case. We can make this cost much lower, amortized over time, by using a technique called path compression.

eqClass(a)
   b = a
   while NEXT(b) not = NULL
       b = NEXT(b)
       toReturn = b
   b = a
   while NEXT(b) not = NULL
       tmp = b
       b = NEXT(b)
       NEXT(tmp) = toReturn
   return toReturn

Before, every time eqClass(a) was called, the path from a to its tree’s sink would have to be found. Now we’ve changed things so that once the path is found the first time, every vertex on the path has its NEXT pointer changed so that next time one gets to the sink at once. In other words eqClass(a) will always cost $\Theta(1)$, except the first time. If we expect eqClass will be called several times for each vertex then, amortizing over time, we could say that the cost of eqClass is ‘almost’ constant.

To illustrate the change, here’s what the above data structure would look like after making the call eqClass(3):

<table>
<thead>
<tr>
<th>NEXT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>NULL</td>
</tr>
<tr>
<td>6</td>
<td>NULL</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>