CS 473: Fundamental Algorithms, Spring 2013

DFS in Directed Graphs, Strong Connected Components, and DAGs

Lecture 2

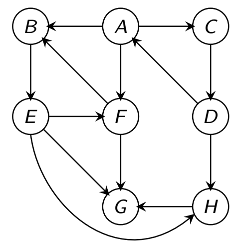
January 19, 2013

Strong Connected Components (s)

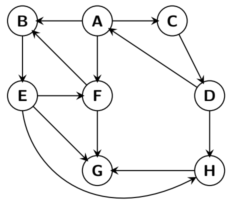
Algorithmic Problem

Find all SCCs of a given directed graph.

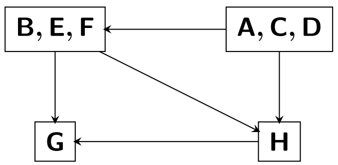
Previous lecture:
Saw an $\mathbf{O(n \cdot (n + m))}$ time algorithm.
This lecture: $\mathbf{O(n + m)}$ time algorithm.



Graph of s



Graph G



Graph of SCCs G^{SCC}

Meta-graph of SCCs

Let $\mathbf{S_1, S_2, \ldots S_k}$ be the strong connected components (i.e., SCCs) of G. The graph of SCCs is G^{SCC}

1. Vertices are $\mathbf{S_1, S_2, \ldots S_k}$
2. There is an edge $\mathbf{(S_i, S_j)}$ if there is some $\mathbf{u} \in \mathbf{S_i}$ and $\mathbf{v} \in \mathbf{S_j}$ such that $\mathbf{(u, v)}$ is an edge in G.

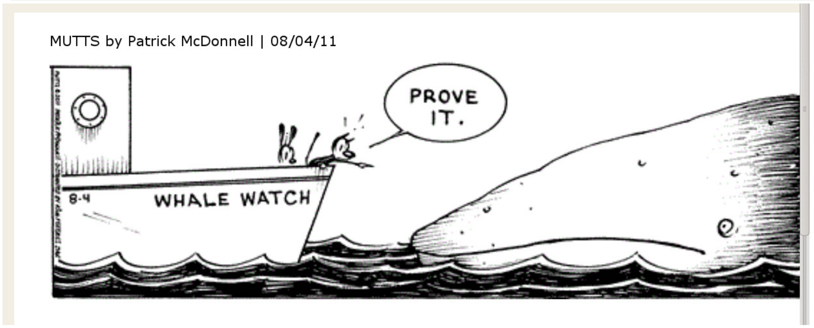
Reversal and SCCs

Proposition

For any graph G, the graph of SCCs of $\mathbf{G^{rev}}$ is the same as the reversal of G^{SCC}.

Proof.

Exercise. □



SCCs and DAGs

Proposition

For any graph G, the graph G^{SCC} has no directed cycle.

Proof.

If G^{SCC} has a cycle $S_1, S_2, \ldots, S_k$ then $S_1 \cup S_2 \cup \cdots \cup S_k$ should be in the same SCC in G. Formal details: exercise. □

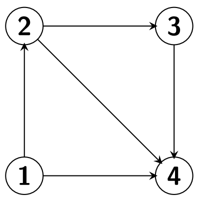
Part I

Directed Acyclic Graphs

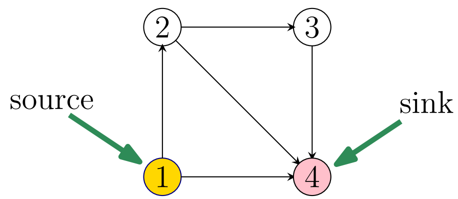
Directed Acyclic Graphs

Definition

A directed graph G is a **directed acyclic graph** (DAG) if there is no directed cycle in G.



Sources and Sinks



Definition

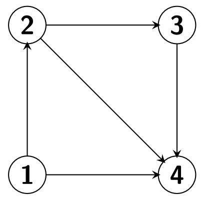
1. A vertex **u** is a **source** if it has no in-coming edges.
2. A vertex **u** is a **sink** if it has no out-going edges.

Simple Properties

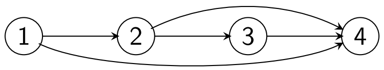
1. Every DAG G has at least one source and at least one sink.
2. If G is a DAG if and only if G^{rev} is a DAG.
3. G is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Topological Ordering/Sorting



Graph G



Topological Ordering of G

Definition

A **topological ordering**/**topological sorting** of $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is an ordering $\prec$ on $\mathbf{V}$ such that if $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}$ then $\mathbf{u} \prec \mathbf{v}$.

Informal equivalent definition:

One can order the vertices of the graph along a line (say the **x**-axis) such that all edges are from left to right.

s and Topological Sort

Lemma

A directed graph G can be topologically ordered iff it is a DAG.

Proof.

$\Longrightarrow$: Suppose G is not a DAG and has a topological ordering $\prec$. G has a cycle $\mathbf{C} = \mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_k}, \mathbf{u_1}$.
Then $\mathbf{u_1} \prec \mathbf{u_2} \prec \ldots \prec \mathbf{u_k} \prec \mathbf{u_1}$!
That is... $\mathbf{u_1} \prec \mathbf{u_1}$.
A contradiction (to $\prec$ being an order).
Not possible to topologically order the vertices. □

s and Topological Sort

Lemma

A directed graph G can be topologically ordered iff it is a DAG.

Continued.

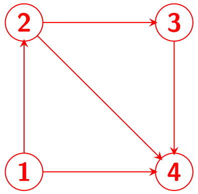
$\Longleftarrow$: Consider the following algorithm:

1. Pick a source **u**, output it.
2. Remove **u** and all edges out of **u**.
3. Repeat until graph is empty.
4. Exercise: prove this gives an ordering.

□

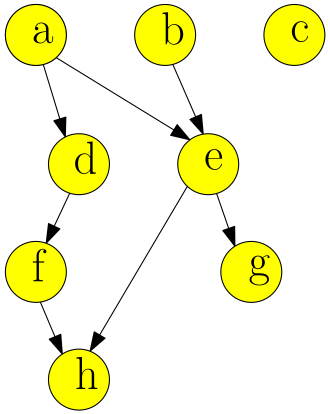
Exercise: show above algorithm can be implemented in $\mathbf{O(m + n)}$ time.

Topological Sort: An Example



Output: 1 2 3 4

Topological Sort: Another Example



s and Topological Sort

Note: A DAG G may have many different topological sorts.

Question: What is a DAG with the most number of distinct topological sorts for a given number **n** of vertices?

Question: What is a DAG with the least number of distinct topological sorts for a given number **n** of vertices?

Using ...

... to check for Acylicity and compute Topological Ordering

Question

Given G, is it a DAG? If it is, generate a topological sort.

DFS based algorithm:

1. Compute **DFS(G)**
2. If there is a back edge then G is not a DAG.
3. Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

Proposition

G is a DAG iff there is no back-edge in **DFS(G)**.

Proposition

If G is a DAG and $\mathrm{post}(\mathbf{v}) > \mathrm{post}(\mathbf{u})$, *then* $(\mathbf{u}, \mathbf{v})$ *is not in G.*

Proof

Proposition

If G is a DAG *and* $\mathrm{post}(\mathbf{v}) > \mathrm{post}(\mathbf{u})$*, then* $(\mathbf{u}, \mathbf{v})$ *is not in G.*

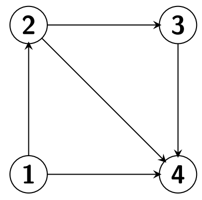
Proof.

Assume $\mathrm{post}(\mathbf{v}) > \mathrm{post}(\mathbf{u})$ *and* $(\mathbf{u}, \mathbf{v})$ is an edge in **G**. We derive a contradiction. One of two cases holds from DFS property.

- Case 1: $[\mathbf{pre}(\mathbf{u}), \mathbf{post}(\mathbf{u})]$ is contained in $[\mathbf{pre}(\mathbf{v}), \mathbf{post}(\mathbf{v})]$. Implies that **u** is explored during **DFS**$(\mathbf{v})$ and hence is a descendent of **v**. Edge $(\mathbf{u}, \mathbf{v})$ implies a cycle in G but G is assumed to be DAG!
- Case 2: $[\mathbf{pre}(\mathbf{u}), \mathbf{post}(\mathbf{u})]$ is disjoint from $[\mathbf{pre}(\mathbf{v}), \mathbf{post}(\mathbf{v})]$. This cannot happen since **v** would be explored from **u**.

□

Example



Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS**$(\mathbf{G})$.

Proof.

If: $(\mathbf{u}, \mathbf{v})$ is a back edge implies there is a cycle **C** consisting of the path from **v** to **u** in **DFS** search tree and the edge $(\mathbf{u}, \mathbf{v})$.

Only if: Suppose there is a cycle $\mathbf{C} = \mathbf{v}_1 \to \mathbf{v}_2 \to \ldots \to \mathbf{v}_k \to \mathbf{v}_1$.
Let $\mathbf{v}_i$ be first node in **C** visited in **DFS**.
All other nodes in **C** are descendants of $\mathbf{v}_i$ since they are reachable from $\mathbf{v}_i$.
Therefore, $(\mathbf{v}_{i-1}, \mathbf{v}_i)$ (or $(\mathbf{v}_k, \mathbf{v}_1)$ if $\mathbf{i} = 1$) is a back edge. □

Topological sorting of a

Input: DAG G. With **n** vertices and **m** edges.

$O(n + m)$ algorithms for topological sorting

(A) Put source **s** of G as first in the order, remove **s**, and repeat. (Implementation not trivial.)

(B) Do **DFS** of G.
Compute post numbers.
Sort vertices by decreasing post number.

Question

How to avoid sorting?
No need to sort - post numbering algorithm can output vertices...

s and Partial Orders

Definition

A **partially ordered set** is a set $\mathbf{S}$ along with a binary relation $\preceq$ such that $\preceq$ is

1. **reflexive** ($\mathbf{a} \preceq \mathbf{a}$ for all $\mathbf{a} \in \mathbf{V}$),
2. **anti-symmetric** ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$ implies $\mathbf{b} \not\preceq \mathbf{a}$), and
3. **transitive** ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{b} \preceq \mathbf{c}$ implies $\mathbf{a} \preceq \mathbf{c}$).

Example: For numbers in the plane define $(\mathbf{x}, \mathbf{y}) \preceq (\mathbf{x}', \mathbf{y}')$ iff $\mathbf{x} \leq \mathbf{x}'$ and $\mathbf{y} \leq \mathbf{y}'$.
Observation: A *finite* partially ordered set is equivalent to a DAG. (No equal elements.)

Observation: A topological sort of a DAG corresponds to a complete (or total) ordering of the underlying partial order.

What's but a sweet old fashioned notion

Who needs a ...

Example

1. $\mathbf{V}$: set of $\mathbf{n}$ products (say, $\mathbf{n}$ different types of tablets).
2. Want to buy one of them, so you do market research...
3. Online reviews compare only pairs of them.
   ...Not everything compared to everything.
4. Given this partial information:
   1. Decide what is the best product.
   2. Decide what is the ordering of products from best to worst.
   3. ...

What DAGs got to do with it?

Or why we should care about DAGs

1. DAGs enable us to represent partial ordering information we have about some set (very common situation in the real world).
2. Questions about DAGs:
   1. Is a graph G a DAG?
      $\iff$
      Is the partial ordering information we have so far is consistent?
   2. Compute a topological ordering of a DAG.
      $\iff$
      Find an a consistent ordering that agrees with our partial information.
   3. Find comparisons to do so DAG has a unique topological sort.
      $\iff$
      Which elements to compare so that we have a consistent ordering of the items.

Part II

Linear time algorithm for finding all strong connected components of a directed graph

Finding all s of a Directed Graph

Problem

Given a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, output *all* its strong connected components.

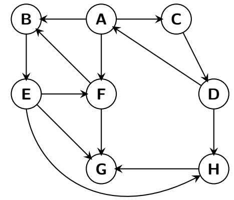
Straightforward algorithm:

```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u): Mark u as visited.
```

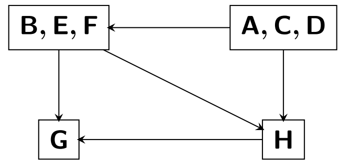
Running time: $\mathbf{O(n(n+m))}$

Is there an $\mathbf{O(n+m)}$ time algorithm?

Structure of a Directed Graph



Graph G



Graph of SCCs G^{SCC}

Reminder

G^{SCC} is created by collapsing every strong connected component to a single vertex.

Proposition

For a directed graph G, its meta-graph G^{SCC} is a DAG.

Linear-time Algorithm for s: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let $\mathbf{u}$ be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**$(\mathbf{u})$ to compute $\mathrm{SCC}(\mathbf{u})$
3. Remove $\mathrm{SCC}(\mathbf{u})$ and repeat

Justification

1. **DFS**$(\mathbf{u})$ only visits vertices (and edges) in $\mathrm{SCC}(\mathbf{u})$
2. ... since there are no edges coming out a sink!
3. **DFS**$(\mathbf{u})$ takes time proportional to size of $\mathrm{SCC}(\mathbf{u})$
4. Therefore, total time $\mathbf{O(n+m)}$!

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

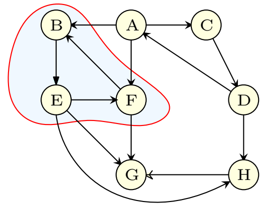
Answer: **DFS(G)** gives some information!

Post-visit times of s

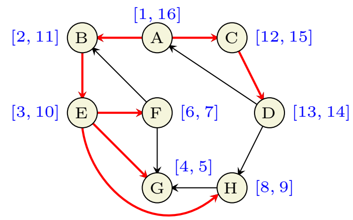
Definition

Given G and a SCC **S** of G, define $\mathrm{post}(\mathbf{S}) = \max_{\mathbf{u} \in \mathbf{S}} \mathrm{post}(\mathbf{u})$ where post numbers are with respect to some **DFS(G)**.

An Example



Graph G



Graph with pre-post times for **DFS(A)**; black edges in tree

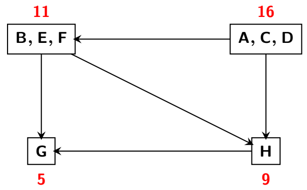


Figure: G^{SCC} with post times

Graph of strong connected components

... and post-visit times

Proposition

If **S** *and* **S′** *are* SCC*s in G and* $(\mathbf{S}, \mathbf{S'})$ *is an edge in* G^{SCC} *then* $\mathrm{post}(\mathbf{S}) > \mathrm{post}(\mathbf{S'})$.

Proof.

Let **u** be first vertex in $\mathbf{S} \cup \mathbf{S'}$ that is visited.

1. If $\mathbf{u} \in \mathbf{S}$ then all of **S′** will be explored before **DFS(u)** completes.
2. If $\mathbf{u} \in \mathbf{S'}$ then all of **S′** will be explored before any of **S**.

□

A False Statement: If **S** and **S′** are SCCs in G and $(\mathbf{S}, \mathbf{S'})$ is an edge in G^{SCC} then for *every* $\mathbf{u} \in \mathbf{S}$ and $\mathbf{u'} \in \mathbf{S'}$, $\mathrm{post}(\mathbf{u}) > \mathrm{post}(\mathbf{u'})$.

Topological ordering of the strong components

Corollary

Ordering SCC*s in decreasing order of* $\mathrm{post}(\mathbf{S})$ *gives a topological ordering of* G^{SCC}

Recall: for a DAG, ordering nodes in decreasing post-visit order gives a topological sort.

So...
DFS(G) gives some information on topological ordering of $\mathbf{G}^{SCC}$!

Finding Sources

Proposition

The vertex $\mathbf{u}$ *with the highest post visit time belongs to a source* SCC *in* G^{SCC}

Proof.

1. $\mathrm{post}(\mathrm{SCC}(\mathbf{u})) = \mathrm{post}(\mathbf{u})$
2. Thus, $\mathrm{post}(\mathrm{SCC}(\mathbf{u}))$ is highest and will be output first in topological ordering of $\mathbf{G}^{\mathrm{SCC}}$.

□

Finding Sinks

Proposition

The vertex $\mathbf{u}$ *with highest post visit time in* **DFS**$(\mathbf{G}^{\mathrm{rev}})$ *belongs to a sink SCC of G.*

Proof.

1. $\mathbf{u}$ belongs to source SCC of $\mathbf{G}^{\mathrm{rev}}$
2. Since graph of SCCs of $\mathbf{G}^{\mathrm{rev}}$ is the reverse of G^{SCC}, $\mathrm{SCC}(\mathbf{u})$ is sink SCC of G. □

Linear Time Algorithm

...for computing the strong connected components in **G**

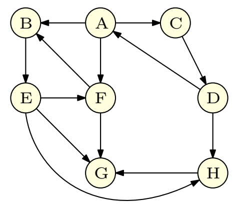
```
do DFS(G^rev) and sort vertices in decreasing post order.
Mark all nodes as unvisited
for each u in the computed order do
    if u is not visited then
        DFS(u)
        Let S_u be the nodes reached by u
        Output S_u as a strong connected component
        Remove S_u from G
```

Analysis

Running time is $O(n + m)$. (Exercise)

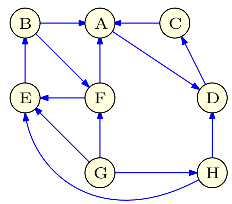
Linear Time Algorithm: An Example - Initial steps

Graph G:



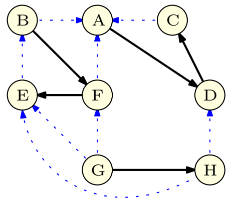
⟹

Reverse graph $\mathbf{G}^{\mathrm{rev}}$:



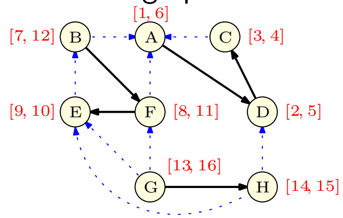
DFS of reverse graph:

⟹



Pre/Post **DFS** numbering of reverse graph:

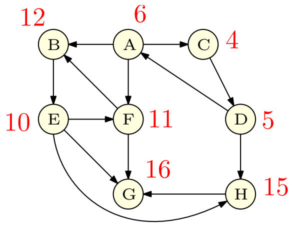
⟹



Linear Time Algorithm: An Example

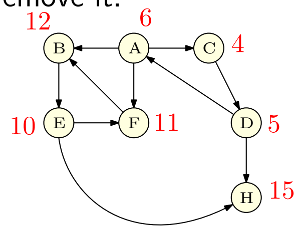
Removing connected components: 1

Original graph G with rev post numbers:



$\Longrightarrow$

Do **DFS** from vertex G remove it.

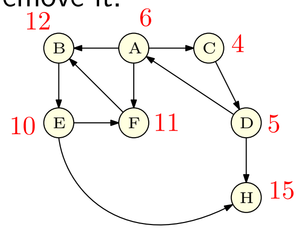


SCC computed:
{G}

Linear Time Algorithm: An Example

Removing connected components: 2

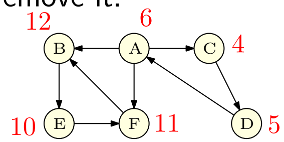
Do **DFS** from vertex G remove it.



SCC computed:
{G}

$\Longrightarrow$

Do **DFS** from vertex **H**, remove it.

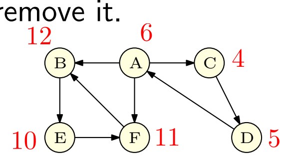


SCC computed:
{G}, {H}

Linear Time Algorithm: An Example

Removing connected components: 3

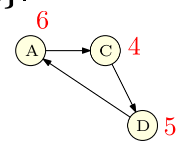
Do **DFS** from vertex **H**, remove it.



SCC computed:
{G}, {H}

$\Longrightarrow$

Do **DFS** from vertex **B**
Remove visited vertices:
{F, B, E}.

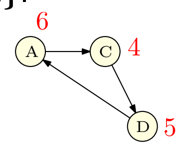


SCC computed:
{G}, {H}, {F, B, E}

Linear Time Algorithm: An Example

Removing connected components: 4

Do **DFS** from vertex **F**
Remove visited vertices:
{F, B, E}.



SCC computed:
{G}, {H}, {F, B, E}

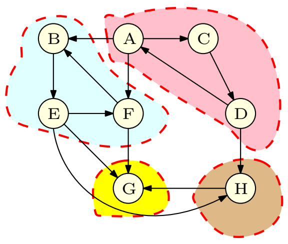
$\Longrightarrow$

Do **DFS** from vertex **A**
Remove visited vertices:
{A, C, D}.

SCC computed:
{G}, {H}, {F, B, E}, {A, C, D}

Linear Time Algorithm: An Example

Final result



SCC computed:
$\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}$
Which is the correct answer!

Obtaining the meta-graph...

Once the strong connected components are computed.

Exercise:

Given all the strong connected components of a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ show that the meta-graph G^{SCC} can be obtained in $\mathbf{O}(\mathbf{m} + \mathbf{n})$ time.

Correctness: more details

1. let $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_k$ be strong components in G
2. Strong components of $\mathbf{G}^{\mathbf{rev}}$ and G are same and meta-graph of G is reverse of meta-graph of $\mathbf{G}^{\mathbf{rev}}$.
3. consider **DFS**($\mathbf{G}^{\mathbf{rev}}$) and let $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k$ be such that $\mathrm{post}(\mathbf{u}_i) = \mathrm{post}(\mathbf{S}_i) = \max_{v \in S_i} \mathrm{post}(\mathbf{v})$.
4. Assume without loss of generality that $\mathbf{post}(\mathbf{u}_k) > \mathbf{post}(\mathbf{u}_{k-1}) \geq \ldots \geq \mathbf{post}(\mathbf{u}_1)$ (renumber otherwise). Then $\mathbf{S}_k, \mathbf{S}_{k-1}, \ldots, \mathbf{S}_1$ is a topological sort of meta-graph of $\mathbf{G}^{\mathbf{rev}}$ and hence $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_k$ is a topological sort of the meta-graph of G.
5. $\mathbf{u}_k$ has highest post number and **DFS**($\mathbf{u}_k$) will explore all of $\mathbf{S}_k$ which is a sink component in G.
6. After $\mathbf{S}_k$ is removed $\mathbf{u}_{k-1}$ has highest post number and **DFS**($\mathbf{u}_{k-1}$) will explore all of $\mathbf{S}_{k-1}$ which is a sink component in remaining graph $\mathbf{G} - \mathbf{S}_k$. Formal proof by induction.

Part III

An Application to `make`

make Utility [Feldman]

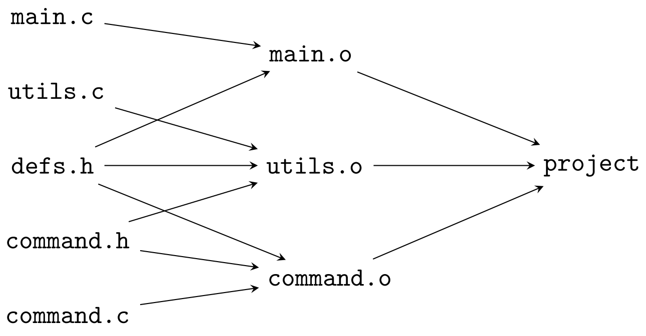
1. Unix utility for automatically building large software applications
2. A makefile specifies
    1. Object files to be created,
    2. Source/object files to be used in creation, and
    3. How to create them

An Example makefile

```
project:  main.o utils.o command.o
    cc -o project main.o utils.o command.o

main.o:  main.c defs.h
    cc -c main.c
utils.o:  utils.c defs.h command.h
    cc -c utils.c
command.o:  command.c defs.h command.h
    cc -c command.c
```


makefile as a Digraph



Computational Problems for make

1. Is the makefile reasonable?
2. If it is reasonable, in what order should the object files be created?
3. If it is not reasonable, provide helpful debugging information.
4. If some file is modified, find the fewest compilations needed to make application consistent.

Algorithms for `make`

1. Is the `makefile` reasonable? Is G a DAG?
2. If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
3. If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
4. If some file is modified, find the fewest compilations needed to make application consistent.
    1. Find all vertices reachable (using **DFS**/**BFS**) from modified files in directed graph, and recompile them in proper order. Verify that one can find the files to recompile and the ordering in linear time.

Take away Points

1. Given a directed graph G, its SCCs and the associated acyclic meta-graph G^{SCC} give a structural decomposition of G that should be kept in mind.
2. There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.
3. DAGs arise in many application and topological sort is a key property in algorithm design. Linear time algorithms to compute a topological sort (there can be many possible orderings so not unique).