# Dependency graphs

## Or: why scheduling is *hard*

Koz Ross

19th October, 2017

**[AUT CSC]**

**AUT Computer Science Club**

# Outline

# A simple recipe: tomato scrambled eggs

# A simple recipe: tomato scrambled eggs

1. Chop tomatoes into small pieces.
2. Break the eggs into a bowl.
3. Beat the eggs.
4. Mix the tomatoes into the eggs.
5. Heat up a frying pan to medium heat.
6. Dump the egg-and-tomato mixture into the frying pan.
7. Cook the mixture until desired consistency.
8. Season with salt and pepper.

# A simple recipe: tomato scrambled eggs

1. Chop tomatoes into small pieces.
2. Break the eggs into a bowl.
3. Beat the eggs.
4. Mix the tomatoes into the eggs.
5. Heat up a frying pan to medium heat.
6. Dump the egg-and-tomato mixture into the frying pan.
7. Cook the mixture until desired consistency.
8. Season with salt and pepper.

- Some tasks can be done simultaneously (e.g. $1$ and $3$)

# A simple recipe: tomato scrambled eggs

1. Chop tomatoes into small pieces.
2. Break the eggs into a bowl.
3. Beat the eggs.
4. Mix the tomatoes into the eggs.
5. Heat up a frying pan to medium heat.
6. Dump the egg-and-tomato mixture into the frying pan.
7. Cook the mixture until desired consistency.
8. Season with salt and pepper.

- Some tasks can be done simultaneously (e.g. $1$ and $3$)
- Other tasks *must* be done in a certain order (e.g. $2$ must happen before $3$)

# A simple recipe: tomato scrambled eggs

1. Chop tomatoes into small pieces.
2. Break the eggs into a bowl.
3. Beat the eggs.
4. Mix the tomatoes into the eggs.
5. Heat up a frying pan to medium heat.
6. Dump the egg-and-tomato mixture into the frying pan.
7. Cook the mixture until desired consistency.
8. Season with salt and pepper.

- Some tasks can be done simultaneously (e.g. $1$ and $3$)
- Other tasks *must* be done in a certain order (e.g. $2$ must happen before $3$)
- These together determine how (and how fast!) we can finish all the tasks

# Why does it matter?

- Knowing how many tasks (and which ones) can be done simultaneously helps us finish faster

# Why does it matter?

- Knowing how many tasks (and which ones) can be done simultaneously helps us finish faster
- Knowing how many additional *processes* we can benefit from (and how long they'll spend idle) lets us use our resources more efficiently

# Why does it matter?

- Knowing how many tasks (and which ones) can be done simultaneously helps us finish faster
- Knowing how many additional *processes* we can benefit from (and how long they'll spend idle) lets us use our resources more efficiently
- Finding the *critical path* gives us a hard bound on how quickly we can finish at all

# Why does it matter?

- Knowing how many tasks (and which ones) can be done simultaneously helps us finish faster
- Knowing how many additional *processes* we can benefit from (and how long they'll spend idle) lets us use our resources more efficiently
- Finding the *critical path* gives us a hard bound on how quickly we can finish at all
- *Especially* important for computers (parallel processing is how we get all of our speed gains these days)

# Why does it matter?

- Knowing how many tasks (and which ones) can be done simultaneously helps us finish faster
- Knowing how many additional *processes* we can benefit from (and how long they'll spend idle) lets us use our resources more efficiently
- Finding the *critical path* gives us a hard bound on how quickly we can finish at all
- *Especially* important for computers (parallel processing is how we get all of our speed gains these days)
- A special (and limited) case of *scheduling* (this acts as a 'difficulty floor' for the more general version)

# The basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k = \{x \in \mathbb{N} \mid x < k\}$ for $k \in \mathbb{N}$.

# The basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k = \{x \in \mathbb{N} \mid x < k\}$ for $k \in \mathbb{N}$. For example:

- $\mathbb{N}_0 = \{\}$ (there are no natural numbers less than $0$)

# The basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k = \{x \in \mathbb{N} \mid x < k\}$ for $k \in \mathbb{N}$. For example:

- $\mathbb{N}_0 = \{\}$ (there are no natural numbers less than $0$)
- $\mathbb{N}_1 = \{0\}$

# The basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k = \{x \in \mathbb{N} \mid x < k\}$ for $k \in \mathbb{N}$. For example:

- $\mathbb{N}_0 = \{\}$ (there are no natural numbers less than $0$)
- $\mathbb{N}_1 = \{0\}$
- $\mathbb{N}_5 = \{0, 1, 2, 3, 4\}$

# The basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k = \{x \in \mathbb{N} \mid x < k\}$ for $k \in \mathbb{N}$. For example:

- $\mathbb{N}_0 = \{\}$ (there are no natural numbers less than $0$)
- $\mathbb{N}_1 = \{0\}$
- $\mathbb{N}_5 = \{0, 1, 2, 3, 4\}$

### Definition

Let $A$ be a set. The *powerset of $A$* is the set

$$P(A) = \{x \mid x \subseteq A\}$$

# The basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k = \{x \in \mathbb{N} \mid x < k\}$ for $k \in \mathbb{N}$. For example:

- $\mathbb{N}_0 = \{\}$ (there are no natural numbers less than $0$)
- $\mathbb{N}_1 = \{0\}$
- $\mathbb{N}_5 = \{0, 1, 2, 3, 4\}$

**Definition**

Let $A$ be a set. The *powerset of $A$* is the set

$$P(A) = \{x \mid x \subseteq A\}$$

For example,
$P(\mathbb{N}_3) = \{\{\}, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$.

# Task list

### Definition

A $k$-*task list* is a function $T_k : \mathbb{N}_k \to P(\mathbb{N}_k)$. We call each $t \in \mathbb{N}_k$ a *task of* $T_k$.

# Task list

**Definition**

A $k$-*task list* is a function $T_k : \mathbb{N}_k \to P(\mathbb{N}_k)$. We call each $t \in \mathbb{N}_k$ a *task of* $T_k$.

Essentially, we number all of the tasks sequentially, and associate them with those tasks that need to be done immediately before them.

# Task list

### Definition

A $k$-*task list* is a function $T_k : \mathbb{N}_k \to P(\mathbb{N}_k)$. We call each $t \in \mathbb{N}_k$ a *task of* $T_k$.

Essentially, we number all of the tasks sequentially, and associate them with those tasks that need to be done immediately before them. For example, if our tasks were:

- Get up (numbered $0$)
- Take a dump (numbered $1$)
- Brush teeth (numbered $2$)

# Task list

### Definition

A $k$-*task list* is a function $T_k : \mathbb{N}_k \to P(\mathbb{N}_k)$. We call each $t \in \mathbb{N}_k$ a *task of* $T_k$.

Essentially, we number all of the tasks sequentially, and associate them with those tasks that need to be done immediately before them. For example, if our tasks were:

- Get up (numbered $0$)
- Take a dump (numbered $1$)
- Brush teeth (numbered $2$)

A possible $3$-task list for that would be $\{(0, \{\}), (1, \{0\}), (2, \{0\})\}$.

# Dependencies

# Dependencies

Throughout, let $T_k$ be a $k$-task list, and let $x, y \in \mathbb{N}_k$.

## Dependencies

Throughout, let $T_k$ be a $k$-task list, and let $x, y \in \mathbb{N}_k$.

**Definition**

$x$ *is a (direct) dependency of* $y$ *in* $T_k$ if $x \in T_k(y)$.

# Dependencies

Throughout, let $T_k$ be a $k$-task list, and let $x, y \in \mathbb{N}_k$.

**Definition**

*$x$ is a (direct) dependency of $y$ in $T_k$* if $x \in T_k(y)$.

**Definition**

Let $d_0, d_1, \ldots, d_n \in \mathbb{N}_k$. $(d_0, d_1, \ldots, d_n)$ is a *dependency chain in $T_k$* if for any $0 \leq i < n$, $d_i$ is a dependency of $d_{i+1}$ in $T_k$.

# Dependencies

Throughout, let $T_k$ be a $k$-task list, and let $x, y \in \mathbb{N}_k$.

### Definition

$x$ *is a (direct) dependency of* $y$ *in* $T_k$ if $x \in T_k(y)$.

### Definition

Let $d_0, d_1, \ldots, d_n \in \mathbb{N}_k$. $(d_0, d_1, \ldots, d_n)$ is a *dependency chain in* $T_k$ if for any $0 \leq i < n$, $d_i$ is a dependency of $d_{i+1}$ in $T_k$.

### Definition

$x$ *is a transitive dependency of* $y$ *in* $T_k$ if there exists a dependency chain in $T_k$ whose first element is $x$ and whose last element is $y$.

# Dependencies

Throughout, let $T_k$ be a $k$-task list, and let $x, y \in \mathbb{N}_k$.

**Definition**
$x$ *is a (direct) dependency of* $y$ *in* $T_k$ if $x \in T_k(y)$.

**Definition**
Let $d_0, d_1, \ldots, d_n \in \mathbb{N}_k$. $(d_0, d_1, \ldots, d_n)$ is a *dependency chain in* $T_k$ if for any $0 \leq i < n$, $d_i$ is a dependency of $d_{i+1}$ in $T_k$.

**Definition**
$x$ *is a transitive dependency of* $y$ *in* $T_k$ if there exists a dependency chain in $T_k$ whose first element is $x$ and whose last element is $y$.

**Definition**
We have a *cyclic dependency between* $x$ *and* $y$ *in* $T_k$ if $x$ is a transitive dependency of $y$ and $y$ is a transitive dependency of $x$.

# Dependencies

Throughout, let $T_k$ be a $k$-task list, and let $x, y \in \mathbb{N}_k$.

### Definition

*$x$ is a (direct) dependency of $y$ in $T_k$* if $x \in T_k(y)$.

### Definition

Let $d_0, d_1, \ldots, d_n \in \mathbb{N}_k$. $(d_0, d_1, \ldots, d_n)$ is a *dependency chain in $T_k$* if for any $0 \leq i < n$, $d_i$ is a dependency of $d_{i+1}$ in $T_k$.

### Definition

*$x$ is a transitive dependency of $y$ in $T_k$* if there exists a dependency chain in $T_k$ whose first element is $x$ and whose last element is $y$.

### Definition

We have a *cyclic dependency between $x$ and $y$ in $T_k$* if $x$ is a transitive dependency of $y$ and $y$ is a transitive dependency of $x$.

Where clear, we will drop references to $T_k$ from now on.

# More on task lists

### Observation

*A $k$-task list with a cyclic dependency is impossible to complete.*

# More on task lists

**Observation**

*A $k$-task list with a cyclic dependency is impossible to complete.*

Let $T_k$ be a $k$-task list without any cyclic dependencies.

# More on task lists

### Observation

*A $k$-task list with a cyclic dependency is impossible to complete.*

Let $T_k$ be a $k$-task list without any cyclic dependencies.

### Definition

The *minimum parallel width of* $T_k$ is the size of the smallest set of tasks where no two members are connected by a dependency chain.

# More on task lists

### Observation

*A $k$-task list with a cyclic dependency is impossible to complete.*

Let $T_k$ be a $k$-task list without any cyclic dependencies.

### Definition

The *minimum parallel width of $T_k$* is the size of the smallest set of tasks where no two members are connected by a dependency chain. We define the *maximum parallel width of $T_k$* analogously.

# More on task lists

### Observation

*A $k$-task list with a cyclic dependency is impossible to complete.*

Let $T_k$ be a $k$-task list without any cyclic dependencies.

### Definition

The *minimum parallel width of* $T_k$ is the size of the smallest set of tasks where no two members are connected by a dependency chain. We define the *maximum parallel width of* $T_k$ analogously.

### Definition

The *critical path of* $T_k$ is the longest dependency chain.

# What we want to know about task lists

# What we want to know about task lists

1. Are there any cyclic dependencies?

# What we want to know about task lists

1. Are there any cyclic dependencies?
2. What are the minimum and maximum parallel widths?

# What we want to know about task lists

1. Are there any cyclic dependencies?
2. What are the minimum and maximum parallel widths?
3. What is the length of the critical path?

# What we want to know about task lists

1. Are there any cyclic dependencies?
2. What are the minimum and maximum parallel widths?
3. What is the length of the critical path?
4. How can we compute all of these things?

# What we want to know about task lists

1. Are there any cyclic dependencies?
2. What are the minimum and maximum parallel widths?
3. What is the length of the critical path?
4. How can we compute all of these things?
5. How efficiently can we compute all of these things?

# Graphs

## Definition

A *(directed) graph* $G = (V, E)$ is a tuple, where $V$ is a set of *vertices*, and $E \subseteq V \times V$ is a set of *edges*.

# Graphs

## Definition

A *(directed) graph* $G = (V, E)$ is a tuple, where $V$ is a set of *vertices*, and $E \subseteq V \times V$ is a set of *edges*.

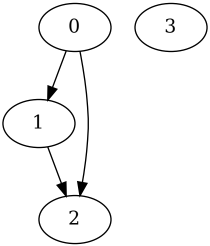
Our graphs will always have $V = \mathbb{N}_k$.

# Graphs

## Definition

A *(directed) graph* $G = (V, E)$ is a tuple, where $V$ is a set of *vertices*, and $E \subseteq V \times V$ is a set of *edges*.

Our graphs will always have $V = \mathbb{N}_k$.




This is a visual representation of the graph $(\mathbb{N}_4, \{(0,1), (0,2), (1,2)\})$.

# Paths and cycles

**Definition**

A *path* in a graph $G = (V, E)$ is a tuple $(p_0, p_1, \ldots, p_n)$, such that:

- Each $p_i \in V$; and
- For any $0 \leq i < n$, $(p_i, p_{i+1}) \in E$.

# Paths and cycles

### Definition

A *path* in a graph $G = (V, E)$ is a tuple $(p_0, p_1, \ldots, p_n)$, such that:

- Each $p_i \in V$; and
- For any $0 \leq i < n$, $(p_i, p_{i+1}) \in E$.

### Definition

A *cycle* is a path whose first and last element are equal.

# Paths and cycles

**Definition**

A *path* in a graph $G = (V, E)$ is a tuple $(p_0, p_1, \ldots, p_n)$, such that:

- Each $p_i \in V$; and
- For any $0 \le i < n$, $(p_i, p_{i+1}) \in E$.

**Definition**

A *cycle* is a path whose first and last element are equal. We say a graph is *cyclic* if it contains any cycles, and *acyclic* otherwise.

# Sources and neighbourhoods

**Definition**

For any edge $e = (a, b)$, we call $a$ the *head of* $e$, and $b$ the *tail of* $e$.

# Sources and neighbourhoods

**Definition**

For any edge $e = (a, b)$, we call $a$ the *head of* $e$, and $b$ the *tail of* $e$.

**Definition**

A vertex $u$ is a *source* if it is not the tail of any edge.

# Sources and neighbourhoods

**Definition**

For any edge $e = (a, b)$, we call $a$ the *head of* $e$, and $b$ the *tail of* $e$.

**Definition**

A vertex $u$ is a *source* if it is not the tail of any edge. A vertex $u$ is a *sink* if it is not the head of any edge.

# Sources and neighbourhoods

### Definition

For any edge $e = (a, b)$, we call $a$ the *head of* $e$, and $b$ the *tail of* $e$.

### Definition

A vertex $u$ is a *source* if it is not the tail of any edge. A vertex $u$ is a *sink* if it is not the head of any edge.

### Definition

Let $G = (V, E)$ be a graph and $u \in V$. The *neighbourhood of* $u$
$N(u) = \{v \in V \mid (u, v) \in E\}$.

# Connecting $k$-task lists and graphs

Let $T_k$ be a $k$-task list. We can represent it as a graph, which we call the *dependency graph of* $T_k$.

# Connecting $k$-task lists and graphs

Let $T_k$ be a $k$-task list. We can represent it as a graph, which we call the *dependency graph of* $T_k$. More specifically:

## Connecting $k$-task lists and graphs

Let $T_k$ be a $k$-task list. We can represent it as a graph, which we call the *dependency graph of $T_k$*. More specifically:

### Definition

The *dependency graph of $T_k$* is $D(T_k) = (V, E)$, such that:

- $V = \mathbb{N}_k$; and
- $E = \{(u, v) \in \mathbb{N}_k^2 \mid u \in T_k(v)\}$

# Connecting $k$-task lists and graphs

Let $T_k$ be a $k$-task list. We can represent it as a graph, which we call the *dependency graph of* $T_k$. More specifically:

**Definition**

The *dependency graph of* $T_k$ is $D(T_k) = (V, E)$, such that:

- $V = \mathbb{N}_k$; and
- $E = \{(u, v) \in \mathbb{N}_k^2 \mid u \in T_k(v)\}$

Any $k$-task list has a unique dependency graph.

# Connecting $k$-task lists and graphs

Let $T_k$ be a $k$-task list. We can represent it as a graph, which we call the *dependency graph of* $T_k$. More specifically:

### Definition

The *dependency graph of* $T_k$ is $D(T_k) = (V, E)$, such that:

- $V = \mathbb{N}_k$; and
- $E = \{(u, v) \in \mathbb{N}_k^2 \mid u \in T_k(v)\}$

Any $k$-task list has a unique dependency graph. This means that we can solve problems for $k$-task lists by solving (related) problems on their dependency graphs.

# Representing dependency graphs

To compute with dependency graphs, we need a way to store them on the computer:

# Representing dependency graphs

To compute with dependency graphs, we need a way to store them on the computer:

**Definition**

Let $G = (\mathbb{N}_k, E)$ be a graph. The *adjacency matrix of* $G$ is a $k \times k$ matrix $M(G)$, such that

$$M(G)[u][v] = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

# Representing dependency graphs

To compute with dependency graphs, we need a way to store them on the computer:

**Definition**

Let $G = (\mathbb{N}_k, E)$ be a graph. The *adjacency matrix of* $G$ is a $k \times k$ matrix $M(G)$, such that

$$M(G)[u][v] = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Consider the previous example graph and its adjacency matrix:

# Representing dependency graphs

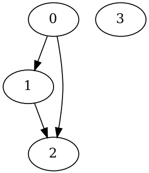
To compute with dependency graphs, we need a way to store them on the computer:

### Definition

Let $G = (\mathbb{N}_k, E)$ be a graph. The *adjacency matrix of* $G$ is a $k \times k$ matrix $M(G)$, such that

$$M(G)[u][v] = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Consider the previous example graph and its adjacency matrix:




$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

# Testing for cyclic dependencies

**Observation**

*If a $k$-task set has a cyclic dependency, its dependency graph will be cyclic.*

# Testing for cyclic dependencies

### Observation

*If a $k$-task set has a cyclic dependency, its dependency graph will be cyclic.*

### Lemma

*If a graph has no source vertices, then it is cyclic.*

# Testing for cyclic dependencies

**Observation**

*If a $k$-task set has a cyclic dependency, its dependency graph will be cyclic.*

**Lemma**

*If a graph has no source vertices, then it is cyclic.*

Thus, we need a way of determining what our source vertices are.

# Finding source vertices

We can check what the source vertices of a graph $G$ are as follows:

# Finding source vertices

We can check what the source vertices of a graph $G$ are as follows:

1. Initially, assume all vertices are sources.

# Finding source vertices

We can check what the source vertices of a graph $G$ are as follows:

1. Initially, assume all vertices are sources.
2. For each row of $M(G)$ and each column, if the $i$th column's entry in that row is $1$, then mark $i$ as not being a source.

# Finding source vertices

We can check what the source vertices of a graph $G$ are as follows:

1. Initially, assume all vertices are sources.
2. For each row of $M(G)$ and each column, if the $i$th column's entry in that row is $1$, then mark $i$ as not being a source.
3. Return the set of all unmarked vertices.

# Finding source vertices

We can check what the source vertices of a graph $G$ are as follows:

1. Initially, assume all vertices are sources.
2. For each row of $M(G)$ and each column, if the $i$th column's entry in that row is $1$, then mark $i$ as not being a source.
3. Return the set of all unmarked vertices.

If $G$ has $n$ vertices, this procedure takes $O(n^2)$ time, as we potentially have to inspect every cell of the adjacency matrix.

# Finding source vertices

We can check what the source vertices of a graph $G$ are as follows:

1. Initially, assume all vertices are sources.
2. For each row of $M(G)$ and each column, if the $i$th column's entry in that row is $1$, then mark $i$ as not being a source.
3. Return the set of all unmarked vertices.

If $G$ has $n$ vertices, this procedure takes $O(n^2)$ time, as we potentially have to inspect every cell of the adjacency matrix.

However, this is not a complete test — a graph with source vertices may still be cyclic!

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

We can use this to design a more thorough method:

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

We can use this to design a more thorough method:

1. Let $S = \emptyset$ and $C$ be the set of sources

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

We can use this to design a more thorough method:

1. Let $S = \emptyset$ and $C$ be the set of sources
2. Repeat until $C$ is empty:

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

We can use this to design a more thorough method:

1. Let $S = \emptyset$ and $C$ be the set of sources
2. Repeat until $C$ is empty:
    2.1 Let $C' = \emptyset$
    2.2 For every $u \in C$:

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

We can use this to design a more thorough method:

1. Let $S = \emptyset$ and $C$ be the set of sources
2. Repeat until $C$ is empty:
    2.1 Let $C' = \emptyset$
    2.2 For every $u \in C$:
        2.2.1 Add $u$ to $S$

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

We can use this to design a more thorough method:

1. Let $S = \emptyset$ and $C$ be the set of sources
2. Repeat until $C$ is empty:
    2.1 Let $C' = \emptyset$
    2.2 For every $u \in C$:
        2.2.1 Add $u$ to $S$
        2.2.2 If any $v \in N(u)$ is in $S$, declare we've found a cycle and stop

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

We can use this to design a more thorough method:

1. Let $S = \emptyset$ and $C$ be the set of sources
2. Repeat until $C$ is empty:
   2.1 Let $C' = \emptyset$
   2.2 For every $u \in C$:
      2.2.1 Add $u$ to $S$
      2.2.2 If any $v \in N(u)$ is in $S$, declare we've found a cycle and stop
      2.2.3 Otherwise, add every $v \in N(u)$ to $C'$

# A more thorough cycle test

## Observation

- *A source cannot be part of any cycle*
- *A cycle must eventually re-visit at least one node*

We can use this to design a more thorough method:

1. Let $S = \emptyset$ and $C$ be the set of sources
2. Repeat until $C$ is empty:
   - 2.1 Let $C' = \emptyset$
   - 2.2 For every $u \in C$:
     - 2.2.1 Add $u$ to $S$
     - 2.2.2 If any $v \in N(u)$ is in $S$, declare we've found a cycle and stop
     - 2.2.3 Otherwise, add every $v \in N(u)$ to $C'$
   - 2.3 Set $C = C'$
3. Declare that there are no cycles and stop.

This process is also $O(n^2)$ — we have to make as many checks as there are edges, and there could be as many as $n^2$.

# Finding parallel widths

When we search for cycles, at the start of each 'outer' loop body, $C$ will contain a set of vertices which are not connected by any dependency chain.

# Finding parallel widths

When we search for cycles, at the start of each 'outer' loop body, $C$ will contain a set of vertices which are not connected by any dependency chain. Thus, the algorithm 'slices' the dependency graph into parallelizable chunks.

# Finding parallel widths

When we search for cycles, at the start of each 'outer' loop body, $C$ will contain a set of vertices which are not connected by any dependency chain. Thus, the algorithm 'slices' the dependency graph into parallelizable chunks. The size of the biggest chunk will be the maximum parallel width; similarly, the size of the smallest will be the minimum parallel width.

# Finding parallel widths

When we search for cycles, at the start of each 'outer' loop body, $C$ will contain a set of vertices which are not connected by any dependency chain. Thus, the algorithm 'slices' the dependency graph into parallelizable chunks. The size of the biggest chunk will be the maximum parallel width; similarly, the size of the smallest will be the minimum parallel width.

We can determine both by simply setting each of these values as equal to the size of the set of sources, then updating it as we go through.

# Finding parallel widths

When we search for cycles, at the start of each 'outer' loop body, $C$ will contain a set of vertices which are not connected by any dependency chain. Thus, the algorithm 'slices' the dependency graph into parallelizable chunks. The size of the biggest chunk will be the maximum parallel width; similarly, the size of the smallest will be the minimum parallel width.

We can determine both by simply setting each of these values as equal to the size of the set of sources, then updating it as we go through. If we find a cycle, these are both $0$ (as we can't do anything).

# Finding parallel widths

When we search for cycles, at the start of each 'outer' loop body, $C$ will contain a set of vertices which are not connected by any dependency chain. Thus, the algorithm 'slices' the dependency graph into parallelizable chunks. The size of the biggest chunk will be the maximum parallel width; similarly, the size of the smallest will be the minimum parallel width.

We can determine both by simply setting each of these values as equal to the size of the set of sources, then updating it as we go through. If we find a cycle, these are both $0$ (as we can't do anything).

Thus, we can find both the minimum and maximum parallel width in $O(n^2)$ time as well.

# Finding the length of the critical path

The number of times that the outer loop runs must be the length of the critical path — otherwise, we would have to do at least one more iteration before we exhaust every vertex.

# Finding the length of the critical path

The number of times that the outer loop runs must be the length of the critical path — otherwise, we would have to do at least one more iteration before we exhaust every vertex. Thus, we can just count the iterations, and report them at the end.

# Finding the length of the critical path

The number of times that the outer loop runs must be the length of the critical path — otherwise, we would have to do at least one more iteration before we exhaust every vertex. Thus, we can just count the iterations, and report them at the end.

If we have a cycle, the length of the critical path is $\infty$ (as we can't ever actually finish the tasks).

# Finding the length of the critical path

The number of times that the outer loop runs must be the length of the critical path — otherwise, we would have to do at least one more iteration before we exhaust every vertex. Thus, we can just count the iterations, and report them at the end.

If we have a cycle, the length of the critical path is $\infty$ (as we can't ever actually finish the tasks).

Thus, we can find the length of the critical path in $O(n^2)$ time as well!

# Finding the length of the critical path

The number of times that the outer loop runs must be the length of the critical path — otherwise, we would have to do at least one more iteration before we exhaust every vertex. Thus, we can just count the iterations, and report them at the end.

If we have a cycle, the length of the critical path is $\infty$ (as we can't ever actually finish the tasks).

Thus, we can find the length of the critical path in $O(n^2)$ time as well! We can potentially combine all of these together to avoid traversing the graph more than once.

# What does this all mean?

- Solving all of these problems requires quadratic time *and* space, which means that it's inherently not easy

# What does this all mean?

- Solving all of these problems requires quadratic time *and* space, which means that it's inherently not easy
- It is possible to be (asymptotically) more space-efficient and (practically) more time-efficient, but doesn't happen often

# What does this all mean?

- Solving all of these problems requires quadratic time *and* space, which means that it's inherently not easy
- It is possible to be (asymptotically) more space-efficient and (practically) more time-efficient, but doesn't happen often
- Shows that more general scheduling (which has *additional* requirements) is going to be *at least* $O(n^2)$ (and is actually much worse)

# What does this all mean?

- Solving all of these problems requires quadratic time *and* space, which means that it's inherently not easy
- It is possible to be (asymptotically) more space-efficient and (practically) more time-efficient, but doesn't happen often
- Shows that more general scheduling (which has *additional* requirements) is going to be *at least* $O(n^2)$ (and is actually much worse)
- Determining the critical path itself is harder

# What does this all mean?

- Solving all of these problems requires quadratic time *and* space, which means that it's inherently not easy
- It is possible to be (asymptotically) more space-efficient and (practically) more time-efficient, but doesn't happen often
- Shows that more general scheduling (which has *additional* requirements) is going to be *at least* $O(n^2)$ (and is actually much worse)
- Determining the critical path itself is harder

However, despite this, we can still use these algorithms in many cases, as long as the number of tasks we're dealing with isn't *too* large.

# Questions?