CS 106X, Lecture 25
Topological Sort

reading:
Programming Abstractions in C++, Chapter 18
Plan For This Week

• Graphs: Topological Sort (HW8)
• Classes: Inheritance and Polymorphism (HW8)
• Sorting Algorithms
Plan For Today

• Topological Sort
  – Kahn’s Algorithm
  – Recursive DFS
  – Spreadsheets

• Announcements

• Learning Goal: understand how to implement topological sort, and why it and dependency graphs are useful.
Plan for Today

- **Topological Sort**
  - Kahn’s Algorithm
  - Recursive DFS
  - Spreadsheets

- Announcements
Graphs So Far

- Graphs as Data Structures
  - Adjacency List
  - Adjacency Matrix
  - Edge List
- Searching
  - DFS
  - BFS
  - Dijkstra’s Algorithm
  - A* Search
Dependency Graphs
How can I parallelize these tasks to finish more quickly?
What step is best to optimize in this production process?
What are valid orderings of these tasks?
What is an effective AI strategy for resource management games (like Civilization)?
In what order should we recompile source files with dependencies?
How can I efficiently package up and transmit web dependencies?
Topological Sort

Which tasks could we complete first?
Topological Sort

Toast Bread
- Butter Toast

Chop Veggies
- Sauté Veggies
- Add Eggs & Cook
  - Plate Food

Prepare Eggs

Which tasks could we complete first?
Topological Sort

Toast Bread

Butter Toast

Chop Veggies

Sauté Veggies

Add Eggs & Cook

Plate Food

Prepare Eggs
Topological Sort

1. Toast Bread
2. Butter Toast
3. Chop Veggies
4. Sauté Veggies
5. Add Eggs & Cook
6. Plate Food
7. Prepare Eggs
Topological Sort

- Toast Bread
- Butter Toast
- Chop Veggies
- Sauté Veggies
- Add Eggs & Cook
- Plate Food
- Prepare Eggs
Topological Sort

- Toast Bread
- Chop Veggies
- Sauté Veggies
- Add Eggs & Cook
- Plate Food
- Prepare Eggs
- Butter Toast

Toast Bread
Chop Veggies
Topological Sort

1. Toast Bread
2. Chop Veggies
3. Sauté Veggies
4. Add Eggs & Cook
5. Plate Food
6. Prepare Eggs
7. Butter Toast
Topological Sort

- Toast Bread
- Butter Toast
- Chop Veggies
- Sauté Veggies
- Add Eggs & Cook
- Plate Food
- Prepare Eggs
Topological Sort

- Toast Bread
- Chop Veggies
- Butter Toast
- Sauté Veggies
- Add Eggs & Cook
- Plate Food
- Prepare Eggs
Topological Sort

Toast Bread

Butter Toast

Chop Veggies

Sauté Veggies

Add Eggs & Cook

Plate Food

Prepare Eggs
Topological Sort

1. Toast Bread
2. Chop Veggies
3. Prepare Eggs
4. Butter Toast
5. Sauté Veggies
6. Add Eggs & Cook
7. Plate Food

Steps:
1. Toast Bread
2. Chop Veggies
3. Prepare Eggs
4. Sauté Veggies
5. Add Eggs & Cook
6. Plate Food
Topological Sort

1. Toast Bread
2. Chop Veggies
3. Sauté Veggies
4. Add Eggs & Cook
5. Plate Food

Steps:
- Toast Bread
- Chop Veggies
- Sauté Veggies
- Add Eggs & Cook
- Plate Food
Topological Sort

1. Toast Bread
2. Chop Veggies
3. Prepare Eggs
4. Butter Toast
5. Sauté Veggies
6. Add Eggs & Cook
7. Plate Food
8. Prepare Eggs
9. Add Eggs & Cook
10. Plate Food

Steps:
- Toast Bread
- Chop Veggies
- Sauté Veggies
- Add Eggs & Cook
- Plate Food
- Prepare Eggs
Topological Sort

Given a directed (acyclic!) graph $G = (V, E)$, a topological sort is a total ordering of $G$'s vertices such that for every edge $(v, w)$ in $E$, vertex $v$ precedes $w$ in the ordering.
Topological Sort

- Toast Bread
- Chop Veggies
- Prepare Eggs
- Butter Toast
- Sauté Veggies
- Add Eggs & Cook
- Plate Food

Steps:
1. Toast Bread → Butter Toast
2. Chop Veggies → Sauté Veggies → Add Eggs & Cook → Plate Food
3. Prepare Eggs
Given a directed graph $G = (V, E)$, a topological sort is a total ordering of $G$'s vertices such that for every edge $(v, w)$ in $E$, vertex $v$ precedes $w$ in the ordering.
Given a directed graph $G = (V, E)$, a topological sort is a total ordering of $G$'s vertices such that for every edge $(v, w)$ in $E$, vertex $v$ precedes $w$ in the ordering.
Topological Sort

1. Toast Bread
2. Chop Veggies
3. Prepare Eggs
4. Butter Toast
5. Sauté Veggies
6. Add Eggs & Cook
7. Plate Food

Make Lunch
Topological Sort: Idea

1. Find node with in-degree of 0
2. Add it at the end of our topological ordering so far, and remove it and its edges from the graph
3. Repeat until no nodes left
Topological Sort Algorithm

ordering := { } // (empty list)

Repeat until graph is empty:
   Find a vertex \( v \) with in-degree of 0
   Delete \( v \) and its outgoing edges from graph
   ordering += v

- This algorithm modifies the passed-in graph 😞
- Have we handled all edge cases?
- What is the runtime of this algorithm?
Topological Sort Algorithm

ordering := { }.  // (empty list)
Repeat until graph is empty:
  Find a vertex v with in-degree of 0
  Delete v and its outgoing edges from graph
ordering += v

- This algorithm modifies the passed-in graph 😞
- Have we handled all edge cases?
- What is the runtime of this algorithm?
Topological Sort Algorithm: Cycles

ordering := { }.
Repeat until graph is empty:
  Find a vertex v with in-degree of 0
  Delete v and its outgoing edges from graph
ordering += v

Task 1
  Task 2
  Task 3
ordering := { }.

Repeat until graph is empty:

Find a vertex \( v \) with in-degree of 0
Delete \( v \) and its outgoing edges from graph

\[
ordering += v
\]
ordering := { }.

Repeat until graph is empty:

Find a vertex \( v \) with in-degree of 0
Delete \( v \) and its outgoing edges from graph

ordering += v

Ordering: { }
Topological Sort Algorithm: Cycles

ordering := { }.
Repeat until graph is empty:
  Find a vertex v with in-degree of 0
  Delete v and its outgoing edges from graph
ordering += v

Ordering: { }
ordering := { }.
Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph
ordering += v

Ordering: { }
ordering := { }.

Repeat until graph is empty:
   Find a vertex v with in-degree of 0
   Delete v and its outgoing edges from graph

ordering += v

Ordering: { “Task 2” }
Topological Sort Algorithm: Cycles

ordering := { }.

Repeat until graph is empty:
  Find a vertex \( v \) with in-degree of 0
  Delete \( v \) and its outgoing edges from graph
ordering += v

Ordering: { “Task 2” }
ordering := { }. Repeat until graph is empty:

Find a vertex $v$ with in-degree of 0
Delete $v$ and its outgoing edges from graph
ordering += $v$

Ordering: { “Task 2” }
ordering := { }.
Repeat until graph is empty:
    Find a vertex \( v \) with in-degree of 0
    - if none, no valid ordering possible
Delete \( v \) and its outgoing edges from graph
ordering += \( v \)

Ordering: { “Task 2” }
ordering := { }.  
Repeat until graph is empty:  
  Find a vertex v with in-degree of 0  
    - if none, no valid ordering possible  
Delete v and its outgoing edges from graph  
ordering += v

• This algorithm modifies the passed-in graph 😞  
• Have we handled all edge cases?  
• What is the runtime of this algorithm?
Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

\[
\text{ordering} := \{ \}\,.
\]

Repeat until graph is empty:

Find a vertex \( v \) with in-degree of 0

- if none, no valid ordering possible

Delete \( v \) and its outgoing edges from graph

\[
\text{ordering} += v
\]
Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

\[ O(1) \]
ordering := { }.

\[ O(V) \]
Repeat until graph is empty:

\[ O(V) \]
Find a vertex v with in-degree of 0
- if none, no valid ordering possible

\[ O(E) \]
Delete v and its outgoing edges from graph

\[ O(1) \]
ordering += v

\[ O(V(V+E)) \]
Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

O(1)  
ordering := { }.

O(V)  
Repeat until graph is empty:

O(V)  
Find a vertex v with in-degree of 0  
- if none, no valid ordering possible

O(E)  
Delete v and its outgoing edges from graph

Is the worst case here really O(E) every time? For example, if one node has all the outgoing edges, then we’ll have an O(E) operation that time, but a no-op every other time.
Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

\[ \text{O}(1) \]
ordering := \{ \}.

\[ \text{O}(V) \]
Repeat until graph is empty:

\[ \text{O}(V) \]
Find a vertex \( v \) with in-degree of 0
   - if none, no valid ordering possible

\[ \text{O}(E) \]
Delete \( v \) and its outgoing edges from graph

**Key Idea:** every edge can be deleted *at most once.*
Therefore, the total runtime of this operation is \( \text{O}(E) \), not \( \text{O}(V*E) \).
Topological Sort Algorithm: Runtime

For graph with V vertexes and E edges:

\[ O(1) \]
\textit{ordering} := \{ \}.

\[ O(V) \]
Repeat until graph is empty:

\[ O(V) \]
Find a vertex \( v \) with in-degree of 0
- if none, no valid ordering possible

\[ O(E) \text{ *total*} \]
Delete \( v \) and its outgoing edges from graph

\[ O(1) \]
\textit{ordering} += \( v \)

Tighter bound: \( O(V^2 + E) \)
Topological Sort Algorithm: Runtime

For graph with \( V \) vertexes and \( E \) edges:

\[
O(1) \\
\text{ordering} := \{ \}. \\
\]

\[
O(V) \\
\text{Repeat until graph is empty:} \\
\]

\[
O(V) \\
\text{Find a vertex } \ v \text{ with in-degree of } 0 \\
- \text{ if none, no valid ordering possible} \\
O(E) \ *\text{total}\* \\
\text{Delete } \ v \text{ and its outgoing edges from graph} \\
O(1) \\
\text{ordering} += v \\
\]

Can we make this more efficient?
Topological Sort: Take 2

orderings := { }.
Repeat until graph is empty:
  Find a vertex v with in-degree of 0
  - if none, no valid ordering possible
Delete v and its outgoing edges from graph
orderings += v
3 Key Ideas for Improvement

- Keep a *queue of nodes with in-degree 0* so we don’t have to search for nodes multiple times.
- A node’s in-degree only changes when one of its prerequisites is completed. Therefore, when completing a task, check if any of its neighbors now has in-degree 0.
- Keep a *map of nodes’ in-degrees* so we don’t need to modify the graph.
Topological Sort: Take 2

0-In-Degree Queue: {} 
In-Degree Map: {}
0-In-Degree Queue: \{ “A”, “D”, “E”, “G” \}

Topological Sort: Take 2

0-In-Degree Queue: { “A”, “D”, “E”, “G” }

Topological Sort: Take 2

0-In-Degree Queue: { “D”, “E”, “G” }
0-In-Degree Queue: { "D", "E", "G", "B" } 
In-Degree Map: { "A":0, "B":0, "C":1, "D":0, "E":0, "F":2, "G":0, "H":3 }
Topological Sort: Take 2

0-In-Degree Queue: {“D”, “E”, “G”, “B”}
Topological Sort: Take 2

0-In-Degree Queue: { “E”, “G”, “B” }
Topological Sort: Take 2

0-In-Degree Queue: { “E”, “G”, “B” }
Topological Sort: Take 2

0-In-Degree Queue: { “E”, “G”, “B” }
Topological Sort: Take 2

0-In-Degree Queue: { “G”, “B” }
0-In-Degree Queue: { “G”, “B”, “F” }  
Topological Sort: Take 2

0-In-Degree Queue: { “G”, “B”, “F” }
Topological Sort: Take 2

0-In-Degree Queue: { “B”, “F” }
Topological Sort: Take 2

0-In-Degree Queue: { “B”, “F” }
Topological Sort: Take 2

0-In-Degree Queue: { “F” }
Topological Sort: Take 2

0-In-Degree Queue: { “F”, “C” }  
In-Degree Map: { “A”:0, “B”:0, “C”:0, “D”:0, “E”: 0, “F”: 0, “G”:0, “H”:1 }
Topological Sort: Take 2

0-In-Degree Queue: { “F”, “C” }
In-Degree Map: { “A”:0, “B”:0, “C”:0, “D”:0, “E”: 0, “F”: 0, “G”:0, “H”:1 }
Topological Sort: Take 2

0-In-Degree Queue: {“C”}
In-Degree Map: {“A”:0, “B”:0, “C”:0, “D”:0, “E”:0, “F”:0, “G”:0, “H”:1}
Topological Sort: Take 2

0-In-Degree Queue: { “C”, “H” }  
In-Degree Map: { “A”:0, “B”:0, “C”:0, “D”:0, “E”:0, “F”:0, “G”:0, “H”:0 }
Topological Sort: Take 2

0-In-Degree Queue: { "C", "H" }
In-Degree Map: { "A":0, "B":0, "C":0, "D":0, "E":0, "F":0, "G":0, "H":0 }
Topological Sort: Take 2

0-In-Degree Queue: { “H” }
In-Degree Map: { “A”:0, “B”:0, “C”:0, “D”:0, “E”: 0, “F”: 0, “G”:0, “H”:0 }
0-In-Degree Queue: { "H" }
In-Degree Map: { "A":0, "B":0, "C":0, "D":0, "E": 0, "F": 0, "G":0, "H":0 }
Topological Sort: Take 2

0-In-Degree Queue: {}
In-Degree Map: {
  "A":0, "B":0, "C":0, "D":0, "E":0, "F":0, "G":0, "H":0
}
Plan for Today

• **Topological Sort**
  – Kahn’s Algorithm
  – Recursive DFS
  – Spreadsheets

• **Announcements**
Kahn’s Algorithm

\( \text{map} := \{ \text{each vertex} \rightarrow \text{its in-degree} \}. \)

\( \text{queue} := \{ \text{all vertices with in-degree} = 0 \}. \)

\( \text{ordering} := \{ \}. \)

Repeat until queue is empty:

- Dequeue the first vertex \( v \) from the queue.
  \( \text{ordering} += v. \)

- Decrease the in-degree of all \( v \)'s neighbors by 1 in the \( \text{map}. \)
  \( \text{queue} += \{ \text{any neighbors whose in-degree is now 0} \}. \)

- This algorithm doesn’t modify the passed-in graph! 😊
- Have we handled all edge cases?
- What is the runtime of this algorithm?
**Kahn’s Algorithm**

\( \text{map} := \{ \text{each vertex} \rightarrow \text{its in-degree} \} \).

\( \text{queue} := \{ \text{all vertices with in-degree} = 0 \} \).

\( \text{ordering} := \{ \} \).

Repeat until queue is empty:

- Dequeue the first vertex \( v \) from the queue.
  
  \( \text{ordering} += v \).

- Decrease the in-degree of all \( v \)'s neighbors by 1 in the \( \text{map} \).

\( \text{queue} += \{ \text{any neighbors whose in-degree is now 0} \} \).

**Remarks:**

- This algorithm doesn’t modify the passed-in graph! 😊
- Have we handled all edge cases?
- What is the runtime of this algorithm?
Kahn’s Algorithm: Cycles

*map* := \{each vertex → its in-degree\}.

*queue* := \{all vertices with in-degree = 0\}.

*ordering* := \{\}.

Repeat until queue is empty:

Dequeue the first vertex *v* from the queue.

*ordering* += *v*.

Decrease the in-degree of all *v*’s neighbors by 1 in the *map*.

*queue* += \{any neighbors whose in-degree is now 0\}.
**Kahn’s Algorithm: Cycles**

\[ map := \{ \text{each vertex} \rightarrow \text{its in-degree} \}. \]
\[ queue := \{ \text{all vertices with in-degree} = 0 \}. \]
\[ ordering := \{ \}. \]

Repeat until queue is empty:

1. Dequeue the first vertex \( v \) from the queue.
2. \( ordering += v. \)
3. Decrease the in-degree of all \( v \)'s neighbors by 1 in the \( map. \)
4. \( queue += \{ \text{any neighbors whose in-degree is now 0} \}. \)

**Example:**

- **0-In-Degree Queue:** \{ “B” \}
- **In-Degree Map:** \{ “A”:1, “B”:0, “C”: 2 \}
- **Ordering:** \{ \}

\[ A \rightarrow B \rightarrow C \]
Kahn’s Algorithm: Cycles

map := \{each vertex $\rightarrow$ its in-degree\}.

queue := \{all vertices with in-degree = 0\}.

ordering := \{\}.

Repeat until queue is empty:

- Dequeue the first vertex $v$ from the queue.
- ordering += $v$.
- Decrease the in-degree of all $v$'s neighbors by 1 in the map.

queue += \{any neighbors whose in-degree is now 0\}.

0-In-Degree Queue: \{ “B” \}

In-Degree Map: \{ A”:1, “B”:0, “C”: 2 \}

Ordering: \{\}
Kahn’s Algorithm: Cycles

`map := {each vertex → its in-degree}.`
`queue := {all vertices with in-degree = 0}.`
`ordering := { }.

Repeat until queue is empty:
- Dequeue the first vertex v from the queue.
  - `ordering += v.`
- Decrease the in-degree of all v's neighbors by 1 in the `map`.
- `queue += {any neighbors whose in-degree is now 0}.`

0-In-Degree Queue: { “B” }
In-Degree Map: { A”:1, “B”:0, “C”: 2 }
Ordering: { “B” }
Kahn’s Algorithm: Cycles

\[\text{map} := \{\text{each vertex } \rightarrow \text{its in-degree}\}. \]
\[\text{queue} := \{\text{all vertices with in-degree } = 0\}. \]
\[\text{ordering} := \{ \}. \]

Repeat until queue is empty:

Dequeue the first vertex \(v\) from the queue.

\[\text{ordering} += v. \]

Decrease the in-degree of all \(v\)'s neighbors by 1 in the map.

\[\text{queue} += \{\text{any neighbors whose in-degree is now } 0\}. \]

0-In-Degree Queue: \{ \}

In-Degree Map: \{ A": 1, "B": 0, "C": 1 \}

Ordering: \{ "B" \}
Kahn’s Algorithm: Cycles

- **map** := {each vertex → its in-degree}.
- **queue** := {all vertices with in-degree = 0}.
- **ordering** := {}.

Repeat until queue is empty:
- Dequeue the first vertex $v$ from the queue.
  - **ordering** += $v$.
- Decrease the in-degree of all $v$'s neighbors by 1 in the **map**.
- **queue** += {any neighbors whose in-degree is now 0}.

0-In-Degree Queue: {}  
In-Degree Map: { A”:1, “B”:0, “C”: 1 }  
Ordering: { “B” }
Kahn’s Algorithm: Cycles

map := \{each vertex → its in-degree\}.
queue := \{all vertices with in-degree = 0\}.
ordering := \{ \}.

Repeat until queue is empty:

- Dequeue the first vertex \( v \) from the queue.
- \( ordering += v \).
- Decrease the in-degree of all \( v \)'s neighbors by 1 in the \( map \).
- queue += \{any neighbors whose in-degree is now 0\}.

0-In-Degree Queue: \{ \}
In-Degree Map: \{ A"::1, "B":0, "C": 1 \}

Ordering: \{ “B” \}
Kahn’s Algorithm: Cycles

\[ map := \{ \text{each vertex} \rightarrow \text{its in-degree} \}. \]

\[ queue := \{ \text{all vertices with in-degree} = 0 \}. \]

\[ ordering := \{ \}. \]

Repeat until queue is empty:

- Dequeue the first vertex \( v \) from the queue.
  
  \[ ordering += v. \]

- Decrease the in-degree of all \( v \)'s neighbors by 1 in the \( map \).

\[ queue += \{ \text{any neighbors whose in-degree is now} \ 0 \}. \]

0-In-Degree Queue: \{ \}

In-Degree Map: \{ “A”:1, “B”:0, “C”: 1 \}

Ordering: \{ “B” \}
Kahn’s Algorithm: Cycles

\[ \text{map} := \{ \text{each vertex } \to \text{ its in-degree} \}. \]
\[ \text{queue} := \{ \text{all vertices with in-degree } = 0 \}. \]
\[ \text{ordering} := \{ \}. \]

Repeat until queue is empty:
  
  Dequeue the first vertex \( v \) from the queue.
  
  \[ \text{ordering} += v. \]

  Decrease the in-degree of all \( v \)'s neighbors by 1 in the \( \text{map} \).
  
  \[ \text{queue} += \{ \text{any neighbors whose in-degree is now } 0 \}. \]

If all vertices processed, success! Otherwise, there is a cycle.

• This algorithm doesn’t modify the passed-in graph! 😊
• Have we handled all edge cases?
• What is the runtime of this algorithm?
Kahn’s Algorithm: Runtime

map := \{each vertex \rightarrow its in-degree\}.

queue := \{all vertices with in-degree = 0\}.

ordering := \{\}\.

Repeat until queue is empty:

- Dequeue the first vertex \(v\) from the queue.
  ordering += \(v\).
- Decrease the in-degree of all \(v\)'s neighbors by 1 in the map.

queue += \{any neighbors whose in-degree is now 0\}.

For graph with \(V\) vertexes and \(E\) edges
Kahn’s Algorithm: Runtime

\(O(V)\)

map := \{each vertex \(\rightarrow\) its in-degree\}.

queue := \{all vertices with in-degree = 0\}.

ordering := \{\}.

\(O(V)\)

Repeat until queue is empty:

\(O(1)\)

Dequeue the first vertex \(v\) from the queue.

ordering += \(v\).

\(O(E)\)

Decrease the in-degree of all \(v\)'s neighbors by 1 in the map.

queue += \{any neighbors whose in-degree is now 0\}.  

For graph with \(V\) vertexes and \(E\) edges

\(O(VE)\)
Kahn’s Algorithm: Runtime

\( O(V) \)
map := \{each vertex \( \rightarrow \) its in-degree\}.

\( O(V) \)
queue := \{all vertices with in-degree = 0\}.

ordering := \{ \}.

\( O(V) \)
Repeat until queue is empty:

\( O(1) \)
Dequeue the first vertex \( v \) from the queue.
ordering += \( v \).

\( O(E) \) *total*
Decrease the in-degree of all \( v \)'s neighbors by 1 in the map.
queue += \{any neighbors whose in-degree is now 0\}.

\( O(V + E) \)
Plan for Today

• **Topological Sort**
  – Kahn’s Algorithm
  – Recursive DFS
  – Spreadsheets

• Announcements
Revisiting Recursive DFS
Revisiting Recursive DFS
Revisiting Recursive DFS

- **Key idea:** visiting node $a$ means we must have visited all nodes that have $a$ as a prerequisite.
Revisiting Recursive DFS

- **Key idea:** visiting node $a$ means we must have visited all nodes that have $a$ as a prerequisite.
- But different starting points may yield different subsets. **Is there a way we can combine them into one complete topological ordering?**
Revisiting Recursive DFS

A D F G E
B C F G E D
C F G E
D F G E
E
F G
G
H
Revisiting Recursive DFS

A D F G E
B C
H
Revisiting Recursive DFS

H B C A D F G E
Revisiting Recursive DFS
Revisiting Recursive DFS
Revisiting Recursive DFS

A  B  C  D  E  F  G  H

Diagram:

- A
- B
- C
- D
- E
- F
- G
- H

Connections:
- A to B
- B to C
- C to D
- D to E
- E to F
- F to G
- G to H
- A to H
Topological Sort: DFS

• function topologicalSortDFS():
  \[ L = \{ \} \text{.} // \text{to store the sorted vertexes} \]
  while graph contains any unmarked vertexes:
  • select any unmarked vertex \( v \).
  • visit \((L, v)\). 

• function visit \((L, v)\):
  if \( v \) is unmarked:
  • for each neighbor \( n \) of \( v \):
    • visit \((L, n)\).
  • mark \( v \).
  • add \( v \) to front of \( L \).
Topological Sort: DFS

- function `topologicalSortDFS()`:
  
  
  \[
  L = \{ \}. \quad // \text{to store the sorted vertexes}
  \]
  
  while graph contains any unmarked vertexes:
  
  • select any unmarked vertex \( v \).
  
  • `visit(L, v)`.

- function `visit(L, v)`:

  if \( v \) has a `temp.mark`, error.

  if \( v \) is unmarked:

  • `temp.mark v`.

  • for each neighbor \( n \) of \( v \):

    • `visit(L, n)`.

  • mark \( v \).

  • add \( v \) to front of \( L \).
Plan for Today

• **Topological Sort**
  – Kahn’s Algorithm
  – Recursive DFS
  – Spreadsheets

• Announcements
Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have dependencies.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>

Topological Sort: Spreadsheets

![Dependency graph showing the relationships between cells A1, A2, A3, B1, B2, and B3.]
Cells with formulas referencing other cells have dependencies.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have **dependencies**.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Topological Sort: Spreadsheets

Cells with formulas referencing other cells have **dependencies**.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>1.166</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Topological Sort: Spreadsheets

Cells with formulas referencing other cells have dependencies.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>1.166</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have **dependencies**.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>1.166</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have dependencies.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>1.166</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Topological Sort: Spreadsheets

Cells with formulas referencing other cells have dependencies.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>1.166</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have dependencies.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>0.777</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have **dependencies**.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=B2/A2</td>
<td></td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have dependencies. We can use topological sort to efficiently update cells!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=B2/A2</td>
<td></td>
</tr>
</tbody>
</table>

B1 → A1 → B2 → A2 → A3 → B3
Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>

**Diagram:**

```
B1  A1  B2  A2  A3  B3
  ▼    ▼    ▲
```

123
Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=B2/A2</td>
<td></td>
</tr>
</tbody>
</table>
Cells with formulas referencing other cells have **dependencies**. We can use topological sort to efficiently update cells!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>=B1*2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>=A1+B1</td>
<td>=A1+5</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>0.777</td>
</tr>
<tr>
<td></td>
<td>=SUM(A1:A2)</td>
<td>=B2/A2</td>
</tr>
</tbody>
</table>
Plan for Today

• Topological Sort
  – Kahn’s Algorithm
  – Recursive DFS
  – Spreadsheets

• Announcements
Announcements

• Homework 8 (Excel) goes out on Wednesday 11/28, is due Friday 12/7. **No late submissions will be accepted.**

• Guest lecture on Hashing next Monday: **Zach!**
Recap

• Topological Sort
  – Kahn’s Algorithm
  – Recursive DFS
  – Spreadsheets

• Announcements

Next time: Classes – Inheritance and Polymorphism