Lecture 18: Topological sort

Part 1: Problem and math basis
Part 2: Algorithm and software solution
Part 1

Problem and mathematical basis
Un dîner en famille.

— Surtout ! ne parlons pas de l’affaire Dreyfus!

... Ils en ont parlé ...
From a given partial order, produce a compatible total order
The problem

From a given partial order, produce a compatible total order

Partial order: ordering constraints between elements of a set, e.g.
- “Remove the dishes before discussing politics”
- “Walk to Üetliberg before lunch”
- “Take your medicine before lunch”
- “Finish lunch before removing dishes"

Total order: sequence including all elements of set

Compatible: the sequence respects all ordering constraints
- Úetliberg, Medicine, Lunch, Dishes, Politics : OK
- Medicine, Úetliberg, Lunch, Dishes, Politics : OK
- Politics, Medicine, Lunch, Dishes, Úetliberg : not OK
Why this is an important example

- Common problem, occurring in many different areas
- Interesting, non-trivial (but not too hard) algorithm
- Illustrates techniques of algorithms, data structures, complexity (big-O), and other topics of last lecture
- Illustrates software engineering techniques
- Illustrates important mathematical concepts: binary relations and in particular order relations
- It’s just beautiful!

Today: problem and math basis
Next time: detailed algorithm and component
Topological sort: example uses

- From a dictionary, produce list of definitions ("glossary") such that no word occurs prior to its definition

- Produce a complete schedule for carrying out a set of tasks, subject to ordering constraints
  (This is done for scheduling maintenance tasks for industrial tasks, often with thousands of constraints)

- Produce a version of a class with features reordered, so that no call to a feature appears before its declaration
Rectangles with overlap constraints

Constraints: \([B, A], [D, A], [A, C], [B, D], [D, C]\)
Displaying rectangles with overlap constraints

Constraints: \([B, A], [D, A], [A, C], [B, D], [D, C]\)

Possible display order:

\[B \quad D \quad E \quad A \quad C\]
An example in EiffelStudio

To implement \( x.f \) with dynamic binding: need routine table

Routines \( \rightarrow \)

Classes (types) \( \downarrow \)

`LINKED_LIST`

Routine pointer

Void
The problem

From a given partial order, produce a compatible total order

Partial order: ordering constraints between elements of a set, e.g.
- “Remove the dishes before discussing politics”
- “Walk to Üetliberg before lunch”
- “Take your medicine before lunch”
- “Finish lunch before removing dishes”

Total order: sequence including all elements of set

Compatible: the sequence respects all ordering constraints
- Üetliberg, Medicine, Lunch, Dishes, Politics: OK
- Medicine, Üetliberg, Lunch, Dishes, Politics: OK
- Politics, Medicine, Lunch, Dishes, Üetliberg: not OK
"Remove the dishes before discussing politics"

"Walk to Üetliberg before lunch"

"Take your medicine before lunch"

"Finish lunch before removing dishes"
Sometimes there is no solution

- "Introducing recursion requires that students know about stacks"
- "You must discuss abstract data types before introducing stacks"
- "Abstract data types rely on recursion"

The constraints introduce a cycle
Given:

- A type $G$
- A set of elements of type $G$
- A set of constraints between these elements

Required:

- An enumeration of the elements, in an order compatible with the constraints

```plaintext
class ORDERABLE[G] feature

  elements: LIST[G]

  constraints: LIST[TUPLE[G, G]]

  topsort: LIST[G]

  ...

  ensure

  compatible(Result, constraints)

end
```
Some mathematical background...
Binary relation on a set

Any property that either holds or doesn’t hold between any two elements of the set

On a set \textit{PERSON} of persons, example relations are:

- \textit{mother} : \(a \text{ mother } b\) holds if and only if \(a\) is the mother of \(b\)
- \textit{father} :
- \textit{child} :
- \textit{sister} :
- \textit{sibling} : (brother or sister)

Notation: \(a \, r \, b\) to express that \(r\) holds of \(a\) and \(b\).
Example: the \textit{before} relation

The set of interest:

\textit{Tasks} = \{\textit{Politics, Lunch, Medicine, Dishes, Üetliberg}\}

The constraining relation:

\textit{Dishes before Politics}

\textit{Üetliberg before Lunch}

\textit{Medicine before Lunch}

\textit{Lunch before Dishes}

“Remove the dishes \textit{before} discussing politics”

“Walk to Üetliberg \textit{before} lunch”

“Take your medicine \textit{before} lunch”

“Finish lunch \textit{before} removing dishes”
Some special relations on a set $X$

**universal** $[X]$: holds between any two elements of $X$

**id** $[X]$: holds between every element of $X$ and itself

**empty** $[X]$: holds between no elements of $X$
Relations: a more precise mathematical view

We consider a relation $r$ on a set $P$ as:

A set of pairs in $P \times P$, containing all the pairs $[x, y]$ such that $x \mathrel{r} y$.

Then $x \mathrel{r} y$ simply means that $[x, y] \in r$

See examples on next slides
A relation is a set: examples

\[
\text{son} = \{[\text{Charles, Elizabeth}], [\text{Charles, Philip}], [\text{William, Charles}], [\text{Harry, Charles}]\}
\]

\[
\text{id} [\mathbb{N}] = \{[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], \ldots\}
\]

\[
\text{universal} [\mathbb{N}] = \{[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], \ldots \\
\quad [1, 0], [1, 1], [1, 2], [1, 3], [1, 4], \ldots \\
\quad [2, 0], [2, 1], [2, 2], [2, 3], [2, 4], \ldots \\
\quad \ldots \}
\]

\[
= \mathbb{N} \times \mathbb{N}
\]

\[
\text{empty} [X] = \emptyset = \{\}
\]
Relations illustrated

- **id** $[\mathbb{N}]$
- **universal** $[\mathbb{N}]$
Example: the *before* relation

“Remove the dishes *before* discussing politics”
“Walk to Üetliberg *before* lunch”
“Take your medicine *before* lunch”
“Finish lunch *before* removing dishes”

The set of interest:

\[ \text{elements} = \{ \text{Politics, Lunch, Medicine, Dishes, Üetliberg} \} \]

The constraining relation:

\[ \text{before} = \{ [\text{Dishes, Politics}], [\text{Üetliberg, Lunch}], [\text{Medicine, Lunch}], [\text{Lunch, Dishes}] \} \]
Using ordinary set operators

\[\text{spouse} = \text{wife} \cup \text{husband}\]

\[\text{sibling} = \text{sister} \cup \text{brother} \cup \text{id} \ [\text{Person}]\]

\[\text{sister} \subseteq \text{sibling}\]

\[\text{father} \subseteq \text{ancestor}\]

\[\text{universal } [X] = X \times X \quad \text{(cartesian product)}\]

\[\text{empty } [X] = \emptyset\]
Possible properties of a relation

(On a set \( X \). All definitions must hold for every \( a, b, c \ldots \in X \).)

- **Total\(^*\):** \((a \neq b) \Rightarrow ((a \text{ r } b) \lor (b \text{ r } a))\)

- **Reflexive:** \(a \text{ r } a\)

- **Irreflexive:** \(\text{not } (a \text{ r } a)\)

- **Symmetric:** \(a \text{ r } b \Rightarrow b \text{ r } a\)

- **Antisymmetric:** \((a \text{ r } b) \land (b \text{ r } a) \Rightarrow a = b\)

- **Asymmetric:** \(\text{not } ((a \text{ r } b) \land (b \text{ r } a))\)

- **Transitive:** \((a \text{ r } b) \land (b \text{ r } c) \Rightarrow a \text{ r } c\)

\(^*\)Definition of “total” is specific to this discussion (there is no standard definition). The other terms are standard.
Examples (on a set of persons)

<table>
<thead>
<tr>
<th>Relation</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>sibling</td>
<td>Reflexive, symmetric, transitive</td>
</tr>
<tr>
<td>sister</td>
<td>Irreflexive</td>
</tr>
<tr>
<td>family_head</td>
<td>Reflexive, antisymmetric</td>
</tr>
</tbody>
</table>

- **Total:** \((a \neq b) \Rightarrow (a \mathcal{R} b) \lor (b \mathcal{R} a)\)
- **Reflexive:** \(a \mathcal{R} a\)
- **Irreflexive:** \(\text{not} (a \mathcal{R} a)\)
- **Symmetric:** \(a \mathcal{R} b \Rightarrow b \mathcal{R} a\)
- **Antisymmetric:** \((a \mathcal{R} b) \land (b \mathcal{R} a) \Rightarrow a = b\)
- **Asymmetric:** \(\text{not} ((a \mathcal{R} b) \land (b \mathcal{R} a))\)
- **Transitive:** \((a \mathcal{R} b) \land (b \mathcal{R} c) \Rightarrow a \mathcal{R} c\)

*(a family_head b means a is the head of b's family, with one head per family)*
Total order relation (strict)

A relation is a **strict total order** if it is:

- Total
- Irreflexive
- Transitive

**Total:**

\[(a \neq b) \Rightarrow (a \mathrel{r} b) \vee (b \mathrel{r} a)\]

**Reflexive:**

\[a \mathrel{r} a\]

**Irreflexive:**

\[\text{not}\ (a \mathrel{r} a)\]

**Symmetric:**

\[a \mathrel{r} b \Rightarrow b \mathrel{r} a\]

**Antisymmetric:**

\[(a \mathrel{r} b) \land (b \mathrel{r} a) \Rightarrow a = b\]

**Asymmetric:**

\[\text{not}\ ((a \mathrel{r} b) \land (b \mathrel{r} a))\]

**Transitive:**

\[(a \mathrel{r} b) \land (b \mathrel{r} c) \Rightarrow a \mathrel{r} c\]

Example: “less than” \(<\) on natural numbers

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

...
Theorem

A strict (total) order is *asymmetric*
A relation is a strict total order if it is:

- Total
- Irreflexive
- Transitive

**Theorem:** A strict total order is *asymmetric*
Total order relation (non-strict)

Relation is non-strict total order if:

- Total
- Reflexive
- Transitive
- Antisymmetric

Example: “less than or equal” ≤ on natural numbers

0 ≤ 0
0 ≤ 1, 1 ≤ 1
0 ≤ 2, 1 ≤ 2, 2 ≤ 2
0 ≤ 3, 1 ≤ 3, 2 ≤ 3
0 ≤ 4, ..., 1 ≤ 4, ..., 2 ≤ 4, ...
Total order relation (strict)

Relation is strict total order if:

- Total
- Irreflexive
- Transitive

Total: \((a \neq b) \Rightarrow (a \, r \, b) \lor (b \, r \, a)\)

Reflexive: \(a \, r \, a\)

Irreflexive: \(\text{not } (a \, r \, a)\)

Symmetric: \(a \, r \, b \Rightarrow b \, r \, a\)

Antisymmetric: \((a \, r \, b) \land (b \, r \, a) \Rightarrow a = b\)

Asymmetric: \(\text{not } ((a \, r \, b) \land (b \, r \, a))\)

Transitive: \((a \, r \, b) \land (b \, r \, c) \Rightarrow a \, r \, c\)
Partial order relation (strict)

Relation is **strict partial order** if:

- **Total**
- **Irreflexive**
- **Transitive**

Example: relation between points in a plane:

\( p \preceq q \) if both:

- \( x_p < x_q \)
- \( y_p < y_q \)
A strict (total) order is asymmetric.

A total order is a partial order.

("partial" order really means possibly partial)
Example partial order

Here the following hold:

- \( a \preceq b \)
- \( c \preceq d \)
- \( a \preceq d \)

No link between \( a \) and \( c \), \( b \) and \( c \):

- e.g. neither \( a \preceq c \) nor \( c \preceq a \)

\( p \preceq q \) if both:
- \( x_p \prec x_q \)
- \( y_p \prec y_q \)
Possible topological sorts

\[a \prec b \quad c \prec d\]
\[a \prec d\]
Topological sort understood

Here the relation $\preceq$ is:

$\{[a, b], [a, d], [c, d]\}$

One of the solutions is:

$a, b, c, d$

We are looking for a total order relation $t$ such that $\preceq \subseteq t$
Final statement of topological sort problem

From a given partial order, produce a compatible total order

where:

A partial order $p$ is compatible with a total order $t$ if and only if

\[ p \subseteq t \]
Is a relation defined by a set of constraints, such as

\[
\text{constraints} = \\
\{ [\text{Dishes, Politics}], [\text{Üetliberg, Lunch}], \\
[\text{Medicine, Lunch}], [\text{Lunch, Dishes}] \}
\]

always a partial order?
Powers and transitive closure of a relation

\[ r^{i+1} = r^i \circ r \] where \( \circ \) is composition

Transitive closure

\[ r^+ = r^1 \cup r^2 \cup \ldots \] always transitive
Reflexive transitive closure

\[ r^0 = \text{id} \quad [X] \quad \text{where } X \text{ is the underlying set} \]

\[ r^{i+1} = r^i \circ r \quad \text{where } \circ \text{ is composition} \]

Transitive closure

\[ r^+ = r^1 \cup r^2 \cup \ldots \quad \text{always transitive} \]

Reflexive transitive closure:

\[ r^* = r^0 \cup r^1 \cup r^2 \cup \ldots \quad \text{always transitive and reflexive} \]
Acyclic relation

A relation $r$ on a set $X$ is acyclic if and only if:

$$r^+ \cap \text{id}[X] = \emptyset$$
Acyclic relations and partial orders

Theorems:

- Any (strict) order relation is acyclic.

- A relation is acyclic if and only if its transitive closure is a (strict) order.

  (Also: if and only if its reflexive transitive closure is a nonstrict partial order)
From constraints to partial orders

The partial order of interest is $\text{before}^+$

$\text{before} = \{(\text{Dishes, Politics}), (\text{Üetliberg, Lunch}), (\text{Medicine, Lunch}), (\text{Lunch, Dishes})\}$
What we have seen

The topological sort problem and its applications
Mathematical background:
- Relations as sets of pairs
- Properties of relations
- Order relations: partial/total, strict/nonstrict
- Transitive, reflexive-transitive closures
- The relation between acyclic and order relations
- The basic idea of topological sort

Next: how to do it for
- Efficient operation: $O(m + n)$ for $m$ constraints & $n$ items
- Good software engineering: effective API
Lecture 18: Topological Sort
Part 2: Algorithm and engineering
Back to software…
The basic algorithm idea

topsort
Overall structure (first try)

**Given:**
- A type $G$
- A set of elements of type $G$
- A relation `constraints` on these elements

**Required:**
- An enumeration of the elements in an order compatible with `constraints`

```ruby
class TOPOLOGICAL_SORTABLE[G]
  feature
    constraints: LINKED_LIST[TUPLE[G, G]]
    elements: LINKED_LIST[G]

  topologically_sorted: LINKED_LIST[G]
  require
    no_cycle(constraints)
  do
    ...
  ensure
    compatible(Result, constraints)
end
```

Overall structure (improved)

Instead of a function `topologically_sorted`, use:

- A procedure `process`.
- An attribute `sorted` (set by `process`), to hold the result.

```plaintext
class
  TOPLOGICAL_SORTED [G]
feature
  constraints : LINKED_LIST [TUPLE [G, G]]
  elements : LINKED_LIST [G]

  sorted : LINKED_LIST [G]

process
  require
    no_cycle (constraints)
  do
    ...
  ensure
    compatible (sorted, constraints)
end
end
```
Non-uniqueness

In general there are several possible solutions

In practice topological sort uses an optimization criterion to choose between possible solutions.
A partial order is acyclic

The \(\preceq\) relation:

- Must be a partial order: no cycle in the transitive closure of *constraints*
- This means there is no circular chain of the form

\[
\begin{align*}
& e_0 \preceq e_1 \preceq \ldots \preceq e_n \preceq e_0 \\
\end{align*}
\]

If there is such a cycle, there exists no solution to the topological sort problem!
In topological sort, we are not given the actual relation \( \preceq \), but a relation \( \text{constraints} \), through a set of pairs such as:

\[
\{[\text{Dishes, Out}], [\text{Museum, Lunch}], [\text{Medicine, Lunch}], [\text{Lunch, Dishes}]\}
\]

The relation of interest is:

\[
\preceq = \text{constraints}^+
\]

\( \preceq \) is acyclic if and only if \( \text{constraints} \) contains no set of pairs

\[
\{[f_0, f_1], [f_1, f_2], \ldots, [f_m, f_0]\}
\]

When such a cycle exists, there can be no total order compatible with \( \text{constraints} \)
Overall structure (reminder)

class
  TOPOLOGICAL_SORTED [G]
feature
  constraints: LINKED_LIST [TUPLE [G, G]]
  elements: LINKED_LIST[G]

  sorted: LINKED_LIST[G]

process
  require
    no_cycle (constraints)
  do
    ...
  ensure
    compatible (sorted, constraints)
end
end
Original assumption

process
require 
\texttt{no\_cycle}\ (\texttt{constraints})
do
...
ensure 
\texttt{compatible}\ (\texttt{sorted}, \texttt{constraints})
end

This assumes there are no cycles in the input

Such an assumption is not enforceable in practice

In particular: finding cycles is essentially as hard as topological sort!
Dealing with cycles

Don’t assume anything; find cycles as byproduct of attempt to do topological sort

The scheme for process becomes:

“Attempt to do topological sort, accounting for possible cycles”

if “Cycles found” then
  “Report cycles”
end
Overall structure (as previously improved)

class
  \textit{TOPOLOGICAL\_SORTED}[G]
feature
  \textit{constraints}: LINKED\_LIST [ TUPLE[G, G] ]
  \textit{elements}: LINKED\_LIST[G]
  \textit{sorted}: LINKED\_LIST[G]

process
  require
    \textit{no\_cycle}(\textit{constraints})
  do
    ...\n  ensure
    \textit{compatible}(\textit{sorted}, \textit{constraints})
end
end
class TOPOLOGICAL_SORTED [G]
feature
  elements : LINKED_LIST [G]
  sorted : LINKED_LIST [G]

process
   require  -- No precondition in this version
   do
      ...
   ensure
      compatible (sorted, constraints)
      "sorted contains all elements not initially involved in a cycle"
end
end
Reminder: the basic algorithm idea
The basic loop scheme

... loop

“Find a member next of elements for which constraints contains no pair of the form \([x, next]\)”

sorted.extend(next)

“Remove next from elements, and remove from constraints any pairs of the form \([next, y]\)”

end
Invariant in our first attempt:

"$\text{constraints}^+\text{ has no cycles}"

Invariant in the revised architecture:

"$\text{constraints}^+\text{ has no cycles other than any that were present originally}"

More generally:

$\text{constraints}^+$ is a subset of the original $\text{constraints}^+$
Terminology

If *constraints* has a pair \([x, y]\), we say that

- \(x\) is a **predecessor** of \(y\)
- \(y\) is a **successor** of \(x\)
Algorithm scheme

process
  do
    from
      create {...} sorted.make
    invariant
      "constraints includes no cycles other than original ones" and
      "sorted is compatible with constraints" and
      “All original elements are in either sorted or elements”
    until
      "Every member of elements has a predecessor”
    loop
      next := “A member of elements with no predecessor”
      sorted.extend(next)
      “Remove next from elements”
      “Remove from constraints all pairs [next, y]”
    variant
      "Size of elements”
  if “No more elements” then
    “Report that topological sort is complete”
  else
    “Report cycle in remaining constraints and elements”
end
end
Implementing the algorithm

We start with these data structures, directly reflecting input data:

- **elements**: LINKED_LIST[G]
- **constraints**: LINKED_LIST[TUPLE[G, G]]

(Number of elements: \( n \)
Number of constraints: \( m \))

Example:

- **elements** = \{a, b, c, d\}
- **constraints** = \{[[a, b], [a, d], [b, d], [c, d]]\}
Data structures 1: original

elements = \{a, b, c, d\}

constraints = \{[a, b], [a, d], [b, d], [c, d]\}

Efficiency: The best we can hope for: $O(m + n)$
Basic operations

\[
\text{process} \quad \text{do}
\]

\[
\begin{align*}
\text{from} & \quad \text{create} \{\ldots\} \text{sorted.make} \\
\text{invariant} & \quad \text{"constraints includes no cycles other than original ones" and} \\
& \quad \text{"sorted is compatible with constraints" and} \\
& \quad \text{"All original elements are in either sorted or elements"} \\
\text{until} & \quad \text{"Every member of elements has a predecessor"} \\
\text{loop} & \quad \text{next := "A member of elements with no predecessor"} \\
& \quad \text{sorted.extend(next)} \\
& \quad \text{"Remove next from elements"} \\
& \quad \text{"Remove from constraints all pairs of the form [next, y]"} \\
\text{variant} & \quad \text{"Size of elements"} \\
\text{end} & \quad \text{if "No more elements" then} \\
& \quad \text{"Report that topological sort is complete"} \\
\text{else} & \quad \text{"Report cycle, in constraints and elements"} \\
\text{end} \\
\end{align*}
\]
The operations we need \((n\text{ times})\)

- Find out if there's any element with no predecessor (and then get one)
- Remove a given element from the set of elements
- Remove from the set of constraints all those starting with a given element
- Find out if there's any element left
Efficiency

The best we can hope for:

$O(m + n)$

(since we have to consider every constraint and every element at least once)
Basic operations

process
  do
    from invariant
      create {...} sorted.make
      "constraints includes no cycles other than original ones" and
      "sorted is compatible with constraints" and
      "All original elements are in either sorted or elements"
    until loop
      "Every member of elements has a predecessor"
      next := "A member of elements with no predecessor"
      sorted.extend (next)
      "Remove next from elements"
      "Remove from constraints all pairs of the form [next, y]"
    variant
      "Size of elements"
    end
  if "No more elements" then
    "Report that topological sort is complete"
  else
    "Report cycle, in constraints and elements"
  end
Data structures 1: original

\[\text{elements} = \{a, b, c, d\}\]
\[\text{constraints} = \{[a, b], [a, d], [b, d], [c, d]\}\]

Efficiency: The best we can hope for:
Using \textit{elements} and \textit{constraints} as given wouldn’t allow us to reach \(O(m + n)\)!
Basic operations

process
  do
    from create {...} sorted.make
    invariant "constraints includes no cycles other than original ones" and "sorted is compatible with constraints" and "All original elements are in either sorted or elements"
    until "Every member of elements has a predecessor"
      loop
        next := "A member of elements with no predecessor"
        sorted.extend(next)
      end
      "Remove next from elements"
      "Remove from constraints all pairs of the form [next, y]"
    end
    variant "Size of elements"
  end
  if "No more elements" then
    "Report that topological sort is complete"
  else
    "Report cycle, in constraints and elements"
  end
end

Basic operations

n times

m times

n times
Implementing the algorithm

Choose a better internal representation

- Give every element a number (allows using arrays)

- Represent constraints in a form adapted to what we want to do with this structure:
  - “Find next such that constraints has no pair of the form [y, next]”
  - “Given next, remove from constraints all pairs of the form [next, y]”
Algorithm scheme (without invariant and variant)

process
  do
    from create {...} sorted.make until
      "Every member of elements has a predecessor"
    loop
      next := “A member of elements with no predecessor”
      sorted.extend(next)
      “Remove next from elements”
      “Remove from constraints all pairs [next, y]”
    end
    if “No more elements” then
      “Report that topological sort is complete”
    else
      “Report cycle in remaining constraints and elements”
    end
  end
Data structure 1: representing \textit{elements}

\textit{elements}: ARRAY[$G$]

\begin{itemize}
  \item Items subject to ordering constraints.
  \item (Replaces the original list)
\end{itemize}

\begin{itemize}
  \item elements = \{a, b, c, d\}
  \item constraints = \{[a, b], [a, d], [b, d], [c, d]\}
\end{itemize}
Data structure 2: representing *constraints*

**successors:** `ARRAY[LINKED_LIST[NUMBER]]`

-- Items that must appear *after* any given one.

```
successors

elements = \{a, b, c, d\}
constraints = \{[a, b], [a, d], [b, d], [c, d]\}
```
Data structure 3: representing *constraints*

**predecessor_count**: ARRAY [INTEGER]

-- Number of items that must appear before a given one.

\[
\begin{array}{c}
4 \\
3 \\
2 \\
1 \\
\end{array} \begin{array}{c}
3 \\
0 \\
1 \\
0 \\
\end{array}
\]

\textit{predecessor\_count}

\textit{elements} = \{a, b, c, d\}

\textit{constraints} = \{[a, b], [a, d], [b, d], [c, d]\}
The basic algorithm idea
Finding a “candidate” (element with no predecessor)

process
do
from create { ... } sorted.make until
  "Every member of elements has a predecessor"
loop
  next := "A member of elements with no predecessor"
  sorted.extend (next)
  "Remove next from elements"
  "Remove from constraints all pairs [next, y]"
end
if "No more elements" then
  "Report that topological sort is complete"
else
  "Report cycle in remaining constraints and elements"
end
Implement

\[ \text{next} := \text{“A member of } \text{elements} \text{ with no predecessors”} \]

as:

Let \textit{next} be an integer, not yet processed, such that

\[ \text{predecessor\_count}[\text{next}] = 0 \]

This requires an \( O(n) \) search through all indexes: bad!

But wait...
Removing successors

```plaintext
process
do
    from create {...} sorted.make until
        "Every member of elements has a predecessor"
    loop
        next := "A member of elements with no predecessor"
        sorted.extend(next)
        "Remove next from elements"
        "Remove from constraints all pairs [next, y]"
    end
    if "No more elements" then
        "Report that topological sort is complete"
    else
        "Report cycle in remaining constraints and elements"
    end
end
```
Removing successors

Implement

“Remove from constraints all pairs \([\text{next}, y]\)”

as a loop over the successors of \(\text{next}\):

\[
\text{targets} := \text{successors}[\text{next}]
\]

\[
\text{from targets.start until targets.after}
\]

\[
\text{loop}
\]

\[
\text{freed} := \text{targets.item}
\]

\[
\text{predecessor_count}[\text{freed}] := \text{predecessor_count}[\text{freed}] - 1
\]

\[
\text{targets.forth}
\]

\[
\text{end}
\]
Removing successors

targets := successors[next]

from targets.start until targets.after

loop

freed := targets.item

predecessor_count[freed] := predecessor_count[freed] - 1

targets.forth

end
Implement "Remove from constraints all pairs [next, y]" as a loop over the successors of next:

```
targets := successors[next]
from targets.start until targets.after
  loop
    freed := targets.item
    predecessor_count[freed] := predecessor_count[freed] - 1
  targets.forth
end
```
Removing successors

\[\text{targets} := \text{successors}[\text{next}] \]

\[\text{from targets.start until targets.after} \]

\[\text{loop} \]

\[\text{freed} := \text{targets.item} \]

\[\text{predecessor\_count}[\text{freed}] := \text{predecessor\_count}[\text{freed}] - 1 \]

\[\text{targets.forth} \]

\[\text{end} \]
Removing successors

```plaintext
targets := successors[next]
from targets.start until targets.after
loop
  freed := targets.item
  predecessor_count[freed] := predecessor_count[freed] − 1
end
```
Algorithm scheme

```
process
  do
    from create {...} sorted.make until
      "Every member of elements has a predecessor"
    loop
      next := "A member of elements with no predecessor"
      sorted.extend(next)
      "Remove next from elements"
      "Remove from constraints all pairs [next, y]"
    end
  if "No more elements" then
    "Report that topological sort is complete"
  else
    "Report cycle in remaining constraints and elements"
  end
end
```
Implement

\[ \text{next} := \text{“A member of elements with no predecessors”} \]

as:

Let \text{next} be an integer, not yet processed, such that \text{predecessor\_count}[\text{next}] = 0

We said:

“Seems to require an \( O(n) \) search through all indexes, but wait...”
Removing successors

\[ \text{targets} := \text{successors}[next] \]
\[ \text{from targets.start until targets.after} \]
\[ \text{loop} \]
\[ \text{freed} := \text{targets.item} \]
\[ \text{predecessor_count}[\text{freed}] := \text{predecessor_count}[\text{freed}] - 1 \]
\[ \text{targets.forth} \]
\[ \text{end} \]
Finding a candidate (2): on the spot

Complement

\[
\text{predecessor\_count}[\text{freed}] := \text{predecessor\_count}[\text{freed}] - 1
\]

by:

\[
\text{if } \text{predecessor\_count}[\text{freed}] = 0 \text{ then}
\]

\[
\text{-- We have found a candidate!}
\]

\[
\text{candidates.put(freed)}
\]

end
Data structure 4: candidates

candidates: STACK[INTEGER]

-- Items with no predecessor

Instead of a stack, candidates can be any dispenser structure, e.g. queue, priority queue

The choice will determine which topological sort we get, when there are several possible ones
process
  do
    from create {...} sorted.make until
      "Every member of elements has a predecessor"
    loop
      next := "A member of elements with no predecessor"
      sorted.extend (next)
      "Remove next from elements"
      "Remove from constraints all pairs [next, y]"
    end
    if "No more elements" then
      "Report that topological sort is complete"
    else
      "Report cycle in remaining constraints and elements"
    end
  end
Finding a candidate (2)

Implement

\[ \text{next} := \text{“A member of } \text{elements} \text{ with no predecessor”} \]

if \( \text{candidates} \) is not empty, as:

\[ \text{next} := \text{candidates.item} \]
Algorithm scheme

process
  do
    from create {...} sorted.make until
      "Every member of elements has a predecessor"
    loop
      next := "A member of elements with no predecessor"
      sorted.extend(next)
      "Remove next from elements"
      "Remove from constraints all pairs [next, y]"
    end
    if "No more elements" then
      "Report that topological sort is complete"
    else
      "Report cycle in remaining constraints and elements"
    end
  end
Finding a candidate (3)

Implement the test

“Every member of *elements* of has a predecessor”

as

not `candidates.is_empty`

To implement the test “No more elements”, keep count of the processed elements and, at the end, compare it with the original number of elements.
Reminder: the operations we need ($n$ times)

- Find out if there's any element with no predecessor (and then get one)
- Remove a given element from the set of elements
- Remove from the set of constraints all those starting with a given element
- Find out if there's any element left
Detecting cycles

\[
\text{process} \\
\text{do} \\
\text{from create } \{\ldots\} \text{ sorted.make } \text{ until} \\
\quad \text{"Every member of } \text{elements} \text{ has a predecessor"} \\
\text{loop} \\
\quad \text{name} := \text{"A member of } \text{elements} \text{ with no predecessor"} \\
\quad \text{sorted.extend}(\text{name}) \\
\quad \text{"Remove } \text{name} \text{ from } \text{elements"} \\
\quad \text{"Remove from } \text{constraints} \text{ all pairs } [\text{name}, y]" \\
\text{end} \\
\text{if} \text{ "No more elements"} \text{ then} \\
\quad \text{"Report that topological sort is complete"} \\
\text{else} \\
\quad \text{"Report cycle in remaining } \text{constraints} \text{ and } \text{elements"} \\
\text{end} \\
\text{end}
\]
Detecting cycles

To implement the test "No more elements", keep count of the processed elements and, at the end, compare it with the original number of elements.
Data structures: summary

elements: ARRAY[G]
    -- Items subject to ordering constraints
    -- (Replaces the original list)

successors: ARRAY[LINKED_LIST[INTEGER]]
    -- Items that must appear after any given one

predecessor_count: ARRAY[INTEGER]
    -- Number of items that must appear before
    -- any given one

candidates: STACK[INTEGER]
    -- Items with no predecessor
Initialization

Must process all elements and constraints to create these data structures

This is $O(m + n)$

So is the rest of the algorithm
Compiling: a useful heuristics

The data structure, in the way it is given, is often not the most appropriate for specific algorithmic processing.

To obtain an efficient algorithm, you may need to turn it into a specially suited form.

We may call this “compiling” the data

Often, the “compilation” (initialization) is as costly as the actual processing, or more, but that’s not a problem if justified by the overall cost decrease.
Another lesson

It may be OK to duplicate information in our data structures:

\[ \text{successors: ARRAY [LINKED_LIST [INTEGER]]} \]

-- Items that must appear after any given one.

\[ \text{predecessor_count: ARRAY [INTEGER]} \]

-- Number of items that must appear before any given one.

This is a simple space-time tradeoff
Key concepts

- A very interesting algorithm, useful in many applications
- Mathematical basis: binary relations
- Remember binary relations & their properties
- Transitive closure, Reflexive transitive closure
- Algorithm: adapting the data structure is the key
- “Compilation” strategy
- Initialization can be as costly as processing
- Algorithm not enough: need API (convenient, extendible, reusable)
- This is the difference between algorithms and software engineering
Software engineering lessons

Great algorithms are not enough

We must provide a solution with a clear interface (API), easy to use

Turn patterns into components