Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Lecture 16: Topological sort
Part 1: Problem and math basis

(Part 2 follows)
Un dîner en famille.

-Surtout! ne parlons pas de l'affaire Dreyfus!

... Ils en ont parlé ...
“Topological sort”

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
- Üetliberg, Medicine, Lunch, Dishes, Politics : OK
- Medicine, Üetliberg, Lunch, Dishes, Politics : OK
- Politics, Medicine, Lunch, Dishes, Üetliberg : not OK
Why we are doing this!

- Very common problem in many different areas
- Interesting, efficient, non-trivial (but not too hard) algorithm
- Illustration of many algorithmic techniques
- Illustration of data structures, complexity (big-Oh notation), and other topics of last lecture
- Illustration of software engineering techniques: from algorithm to component with useful API
- Opportunity to learn or rediscover important mathematical concepts: binary relations (order relations in particular) and their properties
- It’s just beautiful!

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

From a dictionary, produce a list of definitions 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 the features reordered so that no feature call appears before the feature's declaration.
Rectangles with overlap constraints

Constraints: $[B, A], [D, A], [A, C], [B, D], [D, C]$
Rectangles with overlap constraints

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

Possible solution:
An example in EiffelStudio

To implement \texttt{x.f} with dynamic binding: need routine table

\begin{itemize}
\item Routines \rightarrow \textbf{Classes (types)} \downarrow
\item \textbf{LINKED\_LIST}
\end{itemize}
The problem

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

From a given partial order, produce a compatible total order
“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
Overall structure (1)

**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 PERSON of persons, example relations are:

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

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

The set of interest:

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

The constraining relation:

Dishes *before* Politics
Üetliberg *before* Lunch
Medicine *before* Lunch
Lunch *before* Dishes

“Remove the dishes *before* discussing politics”
“Walk to Üetliberg *before* lunch”
“Take your medicine *before* lunch”
“Finish lunch *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 $X$
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 \( x \, r \, y \).

Then \( x \, r \, y \) simply means that \([x, y] \in r\).

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

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

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

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}

empty \([\mathbb{X}]\) = \emptyset = \{\}
Relations illustrated

id \[\mathbb{N}\]

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

The set of interest:

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

The constraining relation:

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

\[
\text{spouse} = \text{wife} \cup \text{husband}
\]
\[
\text{sibling} = \text{sister} \cup \text{brother} \cup \text{id [Person]}
\]
\[
\text{sister} \subseteq \text{sibling}
\]
\[
\text{father} \subseteq \text{ancestor}
\]
\[
\text{universal } [X] = X \times X \quad \text{(cartesian product)}
\]
\[
\text{empty } [X] = \emptyset
\]
Possible properties of a relation

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

- **Total**: $(a \neq b) \Rightarrow ((a \mathrel{r} b) \lor (b \mathrel{r} a))$
- **Reflexive**: $a \mathrel{r} a$
- **Irreflexive**: $\neg (a \mathrel{r} a)$
- **Symmetric**: $a \mathrel{r} b \Rightarrow b \mathrel{r} a$
- **Antisymmetric**: $(a \mathrel{r} b) \land (b \mathrel{r} a) \Rightarrow a = b$
- **Asymmetric**: $\neg ((a \mathrel{r} b) \land (b \mathrel{r} a))$
- **Transitive**: $(a \mathrel{r} b) \land (b \mathrel{r} c) \Rightarrow a \mathrel{r} c$

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

sibling: Reflexive, symmetric, transitive

sister: Irreflexive

family_head: Reflexive, antisymmetric

(a family_head b means a is the head of b's family, with one head per family)

mother: Asymmetric, irreflexive

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

A relation is a strict total order if it is:

- Total
- Irreflexive
- Transitive

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

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

Irreflexive: \(\not(a \, r \, a)\)

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

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

Asymmetric: \(\not((a \, r \, b) \land (b \, r \, a))\)

Transitive: \((a \, r \, b) \land (b \, r \, c) \Rightarrow a \, r \, c\)

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

2 < 3, 2 < 4, ...

...
A strict (total) order is asymmetric
Total order relation (strict)

A relation is a strict total order if it is:

- Total
- Irreflexive
- Transitive

Theorem: A strict total order is asymmetric
Relation is **non-strict total order** if:

- **Total**
- **Reflexive**
- **Transitive**
- **Antisymmetric**

**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**: \((a \, r \, b) \land (b \, r \, c) \Rightarrow a \, r \, c\)

---

**Example**: “less than or equal” \(\leq\) on natural numbers

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0 \leq 0)</td>
<td>(0 \leq 1),</td>
<td>(0 \leq 2),</td>
<td>(0 \leq 3),</td>
<td>(0 \leq 4, \ldots)</td>
<td>(0 \leq 5)</td>
</tr>
<tr>
<td>1</td>
<td>(1 \leq 1)</td>
<td>(0 \leq 2),</td>
<td>(0 \leq 3),</td>
<td>(0 \leq 4, \ldots)</td>
<td>(0 \leq 5)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(2 \leq 2)</td>
<td>(0 \leq 3),</td>
<td>(0 \leq 4, \ldots)</td>
<td>(0 \leq 5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1 \leq 2),</td>
<td>(1 \leq 3),</td>
<td>(1 \leq 4, \ldots)</td>
<td>(1 \leq 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2 \leq 2)</td>
<td>(2 \leq 3),</td>
<td>(2 \leq 4, \ldots)</td>
<td>(2 \leq 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3 \leq 3)</td>
<td>(3 \leq 4, \ldots)</td>
<td>(3 \leq 5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4 \leq 4)</td>
<td>(4 \leq 5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5 \leq 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
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: not \((a \, r \, a)\)

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

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

Asymmetric: 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:

\[ p \preceq q \text{ if both:} \]
\[ x_p < x_q \]
\[ \gamma_p < \gamma_q \]
Theorems

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:

- $a \odot b$
- $c \odot d$
- $a \odot d$

No link between $a$ and $c$, $b$ and $c$:

- e.g. neither $a \odot c$ nor $c \odot a$
Possible topological sorts

\[ a \prec b \quad c \prec d \]

\[ a \odot b \quad c \odot d \]

\[ a \prec d \]
Topological sort understood

Here the relation $\prec$ is:

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

One of the solutions is:

$$a, b, c, d$$

We are looking for a total order relation $\triangleright$ such that $\prec \subseteq \triangleright$
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

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

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

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

Transitive closure

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

\[ r^0 = \text{id}[X] \]

where \( X \) is the underlying set.

where is composition

Transitive closure

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

Reflexive transitive closure:

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

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

$$r^+ \cap \text{id}[X] = \emptyset$$

before $^+$

$\text{id}[X]$
Acyclic relations and partial orders: theorems

1. Any (strict) order relation is acyclic

2. 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)
The partial order of interest is $\text{before}^+$

$\text{before} = 
\{[[\text{Dishes}, \text{Politics}], [\text{Üetliberg}, \text{Lunch}], [\text{Medicine}, \text{Lunch}], [\text{Lunch}, \text{Dishes}]]\}$
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 16: Topological Sort
Part 2: Algorithm and engineering
Back to software...
The basic algorithm idea
Overall structure (original)

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

```plaintext
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
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 
  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
```
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 $e_0 \preceq e_1 \preceq ... \preceq e_n \preceq e_0$

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 $\prec$, 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:

$$\text{constraints}^+$$

is acyclic if and only if $\text{constraints}$ contains no set of pairs

$$\{[f_0, f_1], [f_1, f_2], \ldots, [f_m, f_0]\}$$
Acyclic and partial order relations: theorems (reminder)

1. Any (strict) order relation is acyclic

2. 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)
Acyclic and partial order relations: example (reminder)

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

$\text{before} = \{[[\text{Dishes}, \text{Politics}], [\text{Üetliberg}, \text{Lunch}]], [\text{Medicine, Lunch}], [\text{Lunch, Dishes}]]\}$
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
What about cycles in the input?

The scheme assumed so far:

```plaintext
process
  require no_cycle (constraints)
do
  ...
  ensure compatible (sorted, 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 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
class TOPOLOGICAL_SORTED [G]

feature
  constraints : LINKED_LIST [TUPLE [G, G]]
  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"
Reminder: 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
The loop invariant

Invariant in our first attempt:

“\textit{constraints}^+ \textit{has no cycles}”

Invariant in the revised architecture:

“\textit{constraints}^+ \textit{has no cycles other than any that were present originally}”
If *constraints* has a pair \([x, y]\), we say that

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

```plaintext
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"
    variant
      "Size of 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]"
    end
    if "No more elements" then
      "Report that topological sort is complete"
    else
      "Report cycle in remaining constraints and elements"
    end
  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:

\[
\begin{align*}
\text{elements} & = \{a, b, c, d\} \\
\text{constraints} & = \\
& \{[a, b], [a, d], [b, d], [c, d]\}
\end{align*}
\]
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

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”

variant
“Size of 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 of the form [next, y]”

end
if “No more elements” then
“Report that topological sort is complete”
else
“Report cycle, in constraints and elements”

end
end
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
Data structures 1: original

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

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

Elements:

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

Constraints:

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

Efficiency: The best we can hope for: \(O(m + n)\)

Using elements and constraints as given wouldn’t allow reaching this!
The operations we need ($n$ times)

- Find out if there's any element with no predecessor (then get one)
  \[ O(m) \times n = O(m \times n) \]

- Remove a given element from the set of elements
  \[ O(n) \times n = O(n^2) \]

- Remove from all constraints starting with a given element
  \[ O(m) \times n = O(m \times n) \]

- Find out if there's any element left
  \[ O(1) \times n = O(n) \]

---

**elements**

```
  a  b  c  d
```

**constraints**

```
  a  b  a  d  b  d  c  d
```

$n$ elements  $m$ constraints
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 \([y, \text{next}]\)”
  - “Given **next**, remove from **constraints** all pairs of the form \([\text{next}, y]\)”
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
**Data structure 1: representing elements**

```
elements : ARRAY [G ]

-- Items subject to ordering constraints
-- (Replaces the original list)
```

```
elements = \{a, b, c, d\}
constraints = \{[a, b], [a, d], [b, d], [c, d]\}
```
Data structure 2: representing *constraints*

successors: ARRAY [LINKED_LIST [INTEGER]]

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

```
successors elements = {a, b, c, d}
constraints = {[a, b], [a, d], [b, d], [c, d]}```

```plaintext

4
3
2
1

2  4

4

4```
Data structure 3: representing *constraints*

**predecessor_count**: ARRAY [INTEGER]

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

- predecessor_count

```
4   3
3   0
2   1
1   0
```

elements = {a, b, c, d}

constraints = { [a, b], [a, d], [b, d], [c, d] }
Reminder: basic algorithm idea

topsort
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
  end
Finding a candidate (1)

Implement

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

Let \( next \) be an integer, not yet processed, such that

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

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

But wait…

\[
\begin{align*}
4 & \quad 4 \\
3 & \quad 3 \\
2 & \quad 2 \\
1 & \quad 1 \\
\text{successors} & \\
\text{predecessor\_count} & \\
\end{align*}
\]
Removing successors

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 \(\text{next}\):

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

from \(\text{targets.start}\) until \(\text{targets.after}\)

loop

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

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

\[
\text{targets.forth}
\]

end
Removing successors

\[
\text{targets} := \text{successors} [\text{next} ] \\
\text{from targets.start until} \\
\quad \text{targets.after} \\
\text{loop} \\
\quad \text{freed} := \text{targets.item} \\
\quad \text{predecessor_count [freed]} := \text{predecessor_count [freed]} - 1 \\
\quad \text{targets.forth} \\
\text{end}
\]
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
```
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
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
```
process
  do
    from create {...} sorted.make until
      “Every member of \texttt{elements} has a predecessor”
    loop
      next := “A member of \texttt{elements} with no predecessor”
      sorted.extend (next)
      “Remove \texttt{next} from \texttt{elements}”
      “Remove from \texttt{constraints} all pairs \( [next, y] \)”
    end
    if “No more elements” then
      “Report that topological sort is complete”
    else
      “Report cycle in remaining \texttt{constraints} and \texttt{elements}”
    end
  end
Finding a candidate (1)

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

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

Complement

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

by

\[
\text{if predecessor\_count}[\text{freed}]=0\text{ then}
\]
\[
---\text{We have found a candidate!}
\]
\[
\text{candidates.put}(\text{freed})
\]
\[
\text{end}
\]
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

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

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

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

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
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