Lecture 20: Topological Sort Algorithm
MP1

Decide on footnote ("Info1" or "Intro-course" or whatever)

Michela Pedroni, 9/16/2003
Back to software...
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 $constraints$

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
   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
```

Intro. to Programming, lecture 20: Topological sort algorithm
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 $\prec$ relation:

- Must be a partial order: no cycle in the transitive closure of constraints
- This means there is no circular chain of the form $e_0 \prec e_1 \prec \ldots \prec e_n \prec e_0$

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 \( \preceq \), but a relation 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
\[
\{[f_0, f_1], [f_1, f_2], ..., [f_m, f_0]\}
\]
When such a cycle exists, there can be no total order compatible with \( \text{constraints} \).
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

Intro. to Programming, lecture 20: Topological sort algorithm
process
  require
    \textit{no\_cycle}\,(\textit{constraints})
  do
    ...
  ensure
    \textit{compatible}\,(\textit{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 \textit{process} becomes:

\begin{quote}
"Attempt to do topological sort, accounting for possible cycles"
\end{quote}

\begin{verbatim}
if "Cycles found" then
  "Report cycles"
end
\end{verbatim}
Intro. to Programming, lecture 20: Topological sort algorithm
Intro. to Programming, lecture 20: Topological sort algorithm

Overall structure (final)

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"
    end

end
The basic algorithm idea

Intro. to Programming, lecture 20: Topological sort algorithm
The basic loop scheme

... loop

"Find a member next of elements for which constraints contains no pair of the form \([x, \text{next}]\)"

\textit{sorted.extend}(next)

"Remove next from elements, and remove from constraints any pairs of the form \([\text{next}, y]\)"

end
The loop invariant

Original architecture: “constraints$^+$ has no cycles”

Revised architecture: “constraints$^+$ has no cycles other than any that were present originally”
Overall structure (as previously improved)

```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
```

Intro. to Programming, lecture 20: Topological sort algorithm
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"
end
The loop invariant

Original architecture: “\emph{constraints}^+ \text{ has no cycles}”

Revised architecture: “\emph{constraints}^+ \text{ has no cycles other than any that were present originally}”
Terminology

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

\begin{itemize}
  \item $x$ is a \textbf{predecessor} of $y$
  \item $y$ is a \textbf{successor} of $x$
\end{itemize}
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"

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
```
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**

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

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

\[\text{Efficiency: The best we can hope for: } O(m + n)\]

Intro. to Programming, lecture 20: Topological sort algorithm
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}
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!
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)

```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
```

Intro. to Programming, lecture 20: Topological sort algorithm
Data structure 1: representing *elements*

*elements*: ARRAY[G]

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

$
\begin{array}{c}
4 \\
3 \\
2 \\
1 \\
\end{array}
$

\begin{array}{c}
\text{d} \\
\text{c} \\
\text{b} \\
\text{a} \\
\end{array}

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

\text{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

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

`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

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

predecessor_count

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

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

Intro. to Programming, lecture 20: Topological sort algorithm
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
```
Finding a candidate (1)

Implement

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

as:

Let `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

```
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
```

Intro. to Programming, lecture 20: Topological sort algorithm
Removing successors

Implement

"Remove from constraints all pairs \([next, y]\)"

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

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

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

\[
\text{loop}
\]

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

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

\[
\text{targets.forth}
\]

\[
\text{end}
\]
Removing successors

\[ \text{targets} := \text{successors}[\text{next}] \]
\[ \text{from } \text{targets}.\text{start until} \]
\[ \text{targets}.\text{after} \]
\[ \text{loop} \]
\[ \text{freed} := \text{targets}.\text{item} \]
\[ \text{predecessor\_count}[\text{freed}] := \text{predecessor\_count}[\text{freed}] - 1 \]
\[ \text{targets}.\text{forth} \]
\[ \text{end} \]
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

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

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

\[
\text{loop}
\]

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

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

\[
\text{targets.forth}
\]

\[
\text{end}
\]
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
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 (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 [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
```

Intro. to Programming, lecture 20: Topological sort algorithm
Finding a candidate (2): on the spot

Complement

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

by

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

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

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

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

\textit{elements}: ARRAY[G]
\begin{itemize}
\item -- Items subject to ordering constraints
\item -- (Replaces the original list)
\end{itemize}

\textit{successors}: ARRAY[LINKED\_LIST[INTEGER]]
\begin{itemize}
\item -- Items that must appear after any given one
\end{itemize}

\textit{predecessor\_count}: ARRAY[INTEGER]
\begin{itemize}
\item -- Number of items that must appear before
\item -- any given one
\end{itemize}

\textit{candidates}: STACK[INTEGER]
\begin{itemize}
\item -- Items with no predecessor
\end{itemize}
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:

- **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

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
End of lecture 22