Lecture 20: Topological Sort Algorithm

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

Michela Pedroni, 9/16/2003
Overall structure (improved)

Instead of a function `topologically_sorted`, use:
- A procedure `process`
- An attribute `sorted` (set by `process`), to hold the result.

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 ≤ relation:

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

If there is such a cycle, there exists no solution to the topological sort problem!
Cycles

In topological sort, we are not given the actual relation \( R \), 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:
\[
\mathcal{G} = \text{constraints}
\]
\( \mathcal{G} \) is acyclic if and only if 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 constraints.

Overall structure (reminder)

```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
```

Original assumption

```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)

```java
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
```

Overall structure (final)

```java
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)
  end
```

"sorted contains all elements not initially involved in a cycle"
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, next]"
  sorted.extend(next)
  "Remove next from elements, and remove from constraints any pairs of the form [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)

```
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
```

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 loop invariant

Original architecture: "constraints* has no cycles"

Revised architecture: "constraints* has no cycles other than any that were present originally"
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

\[
\text{process do}
\]

- from create(...) sorted, make invariant
- constraints includes no cycles other than original ones
- sorted is compatible with constraints
- 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 \([next, y]\)”
- 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]]

Example:
- elements = \{a, b, c, d\}
- constraints = \{[a, b], [a, c], [b, d], [c, d]\}
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: $O(m + n)$

---

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 topological sort is complete"
  else
    "Report cycle, in constraints and elements"
  end
end
```

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
Data structures 1: original

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

\[ \text{n elements} \]
\[ \text{m constraints} \]

Efficiency: The best we can hope for: \(O(m + n)\)
Using \(\text{elements}\) and \(\text{constraints}\) as given wouldn’t allow reaching this!

Implementing the algorithm

Choose a better internal representation

- Give every element a number (allows using arrays)
- Represent \(\text{constraints}\) in a form adapted to what we want to do with this structure:
  - “Find next such that \(\text{constraints}\) has no pair of the form \([y, \text{next}]\)”
  - “Given next, remove from \(\text{constraints}\) all pairs of the form \([\text{next}, y]\)”

Algorithm scheme (without invariant and variant)

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

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

\[
\begin{array}{cccc}
4 & d \\
3 & c \\
2 & b \\
1 & a \\
\end{array}
\]

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

Data structure 2: representing constraints

\[
\text{successors: ARRAY[LINKED_LIST[INTEGER]]}
\]
\(\text{--- Items that must appear after any given one.}\)

\[
\begin{array}{cccc}
4 & 4 \\
3 & 4 \\
2 & 2 \\
1 & 4 \\
\end{array}
\]

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

Data structure 3: representing constraints

\[
\text{predecessor_count: ARRAY[INTEGER]}
\]
\(\text{--- Number of items that must appear before a given one.}\)

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

\[
\text{predecessor_count = \{0, 0, 1, 0\}}
\]
\[
\text{constraints = \{[a, b], [a, d], [b, d], [c, d]\}}
\]
Reminder: basic algorithm idea

Finding a "candidate" (element with no predecessor)

```plaintext
process
  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 := "A member of elements with no predecessors"
```

as:

```
let next be an integer, not yet processed, such that
predecessor_count [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)
    end
    if "No more elements" then
      "Report that topological sort is complete"
    else
      "Report cycle in remaining constraints and elements"
    end
  end
end

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

as a loop over the successors of next:

\[
\begin{array}{cccc}
  & 4 & 3 & 2 & 1 \\
4 & o & o & o & x \\
3 & o & o & o & o \\
2 & o & o & o & o \\
1 & o & o & o & o \\
\end{array}
\]

\[\text{predecessor\_count}\]

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

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
```

Implement "Remove from constraints all pairs \([next, y]\)" as a loop over the successors of next.
```
Algorithm scheme

process
  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 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 := "A member of elements with no predecessors"
```

as:

```
Let next be an integer, not yet processed,
such that 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
```
Finding a candidate (2): on the spot

Complement

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

by

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

--- We have found a candidate!

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

Data structure 4: candidates

\[
\text{candidates} : \text{STACK}[\text{INTEGER}]
\]

-- 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

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

\[
\text{end}
\]
Finding a candidate (2)

Implement

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

if candidates is not empty, as:

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

Algorithm scheme

\[
\begin{align*}
\text{process} & \quad \text{do} \\
& \quad \text{from create (...) sorted, make until} \\
& \quad \text{do} \\
& \quad \text{loop} \\
& \quad \text{next} := \text{"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 \( (next, y) \)"} \\
& \quad \text{end} \\
& \quad \text{if \"No more elements\" then} \\
& \quad \text{"Report that topological sort is complete"} \\
& \quad \text{else} \\
& \quad \text{"Report cycle in remaining constraints and elements"} \\
& \quad \text{end} \\
& \quad \text{end} \\
& \quad \text{end} \\
\end{align*}
\]

Finding a candidate (3)

Implement the test

\[ \text{"Every member of elements of has a predecessor"} \]

as

\[ \text{not candidates.is_empty} \]

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)

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

Detecting cycles

```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
```

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[INTERGER]]
  -- Items that must appear after any given one

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

- **candidates**: STACK[INTERGER]
  -- 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:

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