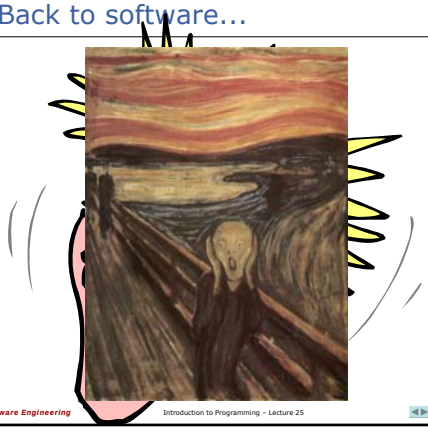
Introduction to Programming

Bertrand Meyer

Last revised 25 January 2005

Lecture 25:
Topological Sort — 2: Algorithm

Back to software...



Overall structure (1)

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] is
        require
            no_cycle (constraints)
        do
            ...
        ensure
            compatible (Result, constraints)
        end
end
```


Overall structure (2)

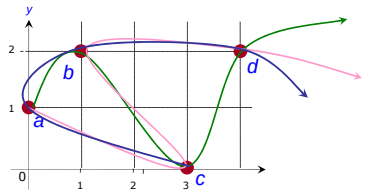
Instead of a function *topologically_sorted*, use:

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

```
class
    TOPOLOGICAL_SORTER [G]
feature
    constraints: LINKED_LIST [TUPLE [G, G]]
    elements: LINKED_LIST [G]
    sorted: LINKED_LIST [G]
    process is
        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.

Cycles

7

- $\prec$ must be a partial order: no cycle in the transitive closure of ***constraints***
  - No circular chain of the form $e_0 \rightarrow e_1, \ldots e_n \rightarrow e_0$
- If there are cycles there exists no solution to the topological sort problem!

Overall structure (2)

8

```
class
    TOPOLOGICAL_SORTER [G]
feature

    constraints: LINKED_LIST [TUPLE [G, G]]

    elements: LINKED_LIST [G]

    sorted: LINKED_LIST [G]

    process is
        require
            no_cycle (constraints)
        do
            ...
        ensure
            compatible (sorted, constraints)
        end
end
```


Cycles

9

- The relation $constraints^+$ must be a partial order: no cycle
- In terms of the original relation this means that *constraints* contains no set of pairs

$$\{[e_0, e_1], [e_1, e_2], \ldots, [e_n, e_0]\}$$

With such a cycle, there exists no total order compatible with *constraints*

Overall structure (2)

10

```
process is
    require
        no_cycle (constraints)
    do
        ...
    ensure
        compatible (sorted, constraints)
    end
```

Assumes there are no cycles in the constraints.

➡ Not realistic! Input may contain errors!

Overall structure (3)

11

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

The task of *process* becomes:

```
"Attempt to do topological sort,
accounting for possible cycles"

if "Cycles found" then
        "Report cycles"
end
```


Overall structure (2)

12

```
process is
    require
        no_cycle (constraints)
    do
        ...
    ensure
        compatible (sorted, constraints)
    end
```

Overall structure (3)


```
process is

        -- No precondition
    do
        ...
    ensure
        compatible (sorted, constraints)

        "sorted contains all elements that were
        not initially involved in a cycle"
    end
```


The basic loop idea


```
...
loop

    "Find a member next of elements for which constraints
    contains no pair of the form [x, next]"

    sorted.extend (next)

    "Remove next from element, and remove from constraints
    any pairs of the form [next, y]"

end
```


Loop invariant

- Scheme 2: "*constraints*$^+$ has no cycles"
- Scheme 3: "*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

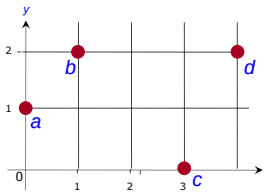

```
process is
    do
        from
            create {...} sorted.make
        invariant
            "constraints includes no cycles other than original ones" and
            "sorted  is compatible with constraints" and
            "All original elements are in either sorted or elements"
        variant
            "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,  given by remaining constraints, in remaining 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*)

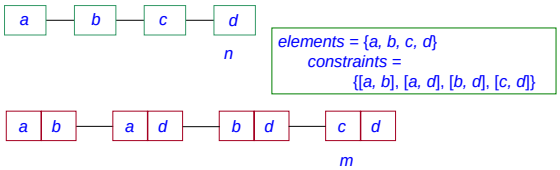


Example:

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

Data structures 1: original

19



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

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

Algorithm scheme

20

```
process is
    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,  given by remaining constraints, in remaining elements"
        end
    end
```


The operations we need (*n* times)

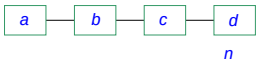
21

- 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

22

elements



n

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

constraints

m

Efficiency: the best we can hope for: **O** (*m*+*n*)

Using *elements* and *constraints* as given wouldn't allow reaching this!

Implementing the algorithm

23

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

24

```
process is
    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,  given by remaining constraints, in remaining elements"
        end
    end
```

Data structure 1: representing *elements*

25

elements: ARRAY [*G*]

-- Items subject to ordering constraints

-- (Replaces the original list)

4	*d*
3	*c*
2	*b*
1	*a*

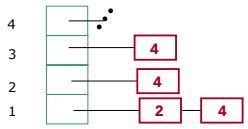
elements = {*a, b, c, d*}
constraints =
{[*a, b*], [*a, d*], [*b, d*], [*c, d*]}

Data structure 2: representing *constraints*

26

successors: ARRAY [*LINKED_LIST* [*INTEGER*]]

-- Items that must appear after any given one



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

Data structure 3: representing *constraints*

27

predecessor_count: ARRAY [*INTEGER*]

-- Number of items that must appear before

-- any given one

4	3
3	0
2	1
1	0

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

Algorithm scheme

28

```
process is
    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,  given by remaining constraints, in remaining elements"
        end
    end
```


Finding a candidate (1)

29

Implement

next := "A member of *elements* with no predecessors"

as:

Let *next* be an integer, not yet processed, such that
predecessor_count.item (*next*) = 0

Seems to require an **O** (*n*) search through all indexes, but wait...

Algorithm scheme

30

```
process is
    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,  given by remaining constraints, in remaining elements"
        end
    end
```

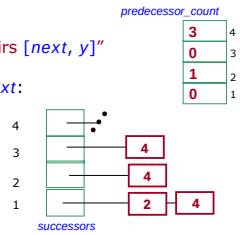
Removing successors

31

Implement

"Remove from *constraints* all pairs [*next*, *y*]"

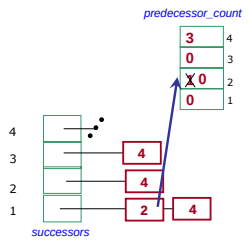
as a loop over the successors of *next*:




```
targets := successors.item (next)
from targets.start until
   targets.after
loop
    freed := targets.item
    remaining := predecessor_count.item (freed)
    predecessor_count.put (remaining −1, freed)
    targets.forth
end
```


Removing successors

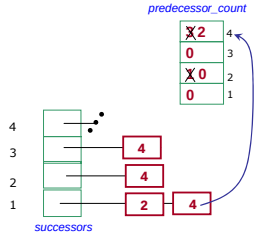
32




```
targets := successors.item (next)
from targets.start until
   targets.after
loop
    freed := targets.item
    remaining := predecessor_count.item (freed)
    predecessor_count.put (remaining −1, freed)
    targets.forth
end
```


Removing successors

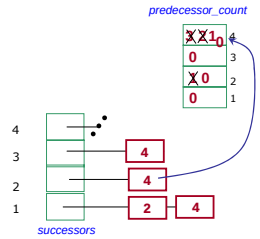
33




```
targets := successors.item (next)
from targets.start until
   targets.after
loop
    freed := targets.item
    remaining := predecessor_count.item (freed)
    predecessor_count.put (remaining −1, freed)
    targets.forth
end
```

Removing successors

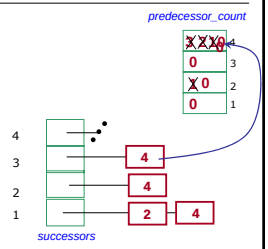
34




```
targets := successors.item (next)
from targets.start until
   targets.after
loop
    freed := targets.item
     remaining := predecessor_count.item (freed)
     predecessor_count.put (remaining −1, freed)
     targets.forth
end
```


Removing successors

35




```
targets := successors.item (next)
from targets.start until
   targets.after
loop
    freed := targets.item
     remaining := predecessor_count.item (freed)
     predecessor_count.put (remaining −1, freed)
     targets.forth
end
```


Finding a candidate (1)

36

Implement

next := "A member of *elements* with no predecessors"

as:

Let *next* be an integer, not yet processed, such that
predecessor_count.item (*next*) = 0

We said:
"Seems to require an **O** (*n*) search through all indexes, but wait..."

Finding a candidate (2): on the spot

37

Complement

```
remaining := predecessor_count.item (freed)
predecessor_count.put (remaining −1, freed)
```

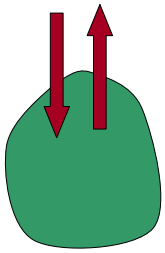
by

```
if remaining = 0 then
        -- We have found a candidate!
    candidates.put (freed)
end
```


Data structure 4: candidates

38

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



Can be any dispenser structure: stack, queue, priority queue

The choice will determine which topological sort we get, when there are several possible ones

Finding a candidate (2)

39

Implement

"Let *next* be a member of *elements* such that *constraints* has no pair of the form [*y*, *next*]"

if *candidates* is not empty, as:

```
next := candidates.item
```

The operations we need

40

- 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

41

```
process is
    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,  given by remaining constraints, in remaining elements"
        end
    end
```


Finding a candidate (3)

42

Implement the test

"Every member 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.

Data structures: summary

43

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

44

- 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

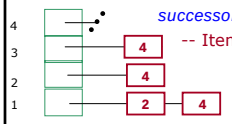
45

- 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

46

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

Software engineering lessons

47

- Great algorithms are not enough
- We must provide a solution with a clear interface (API), easy to use
- Turn patterns into components

48

End of lecture 25