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

class TOPLOGICAL_SORTABLE[$G$]
feature
  constraints : LINKED_LIST [TUPLE[$G$, $G$]]
  elements : LINKED_LIST[$G$]
  topologically_sorted : LINKED_LIST[$G$]

  no_cycle (constraints) do
    compatible (Result, 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 relation:

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

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, but a relation constraints, through a set of pairs such as 

\{(\text{Dishes}, \text{Out}), (\text{Museum}, \text{Lunch}), (\text{Medicine}, \text{Lunch}), (\text{Lunch}, \text{Dishes})\}

The relation of interest is:

\[ \otimes = \text{constraints}^* \]

\(\otimes\) 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)

class TOPOLOGICAL_SORTED \(\mathcal{G}\)

feature

constraints : LINKED_LIST \{ TUPLE \([\mathcal{G}, \mathcal{G}]\) \}

elements : LINKED_LIST\([\mathcal{G}]\)

sorted : LINKED_LIST\([\mathcal{G}]\)

process is

require no_cycle (constraints)

do ...

ensure compatible (sorted, constraints)

end

Original assumption

process is

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 is
    require no_cycle (constraints)
    do
      ensure compatible (sorted, constraints)
    end
  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 is
    require -- No precondition in this version
    do
      ensure compatible (sorted, constraints)
      "sorted contains all elements not initially involved in a cycle"
    end
  end
end
```
The basic algorithm idea

The basic loop scheme

The loop invariant
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 [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]]

Example:
- elements = \([a, b, c, d]\)
- constraints = \([[a, b], [a, d], [b, d], [c, d]]\)
Data structures 1: original

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

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

Basic operations

```
process is
  do
    from create (...) sort, 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"
      no "A member of elements with no predecessor"
      "Sorted, extend (next)"
      "Remove next from elements"
      "Remove from constraints all pairs of the form [next, y]"
    loop
      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 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)

```
process is
do from create \(\ldots\) 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)
```

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

```
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
  4
  3
  2
  1
```

```
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.
```

```
predecessor_count
  4
  3
  2
  1
```

```
elements = {a, b, c, d}
constraints = {{a, b}, {a, d}, {b, d}, {c, d}}
```
Reminder: basic algorithm idea

Finding a "candidate" (element with no predecessor)

Finding a candidate (1)
Removing successors

```
process is do
from create (\ldots) 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"
  end
  if "No more elements" then
    "Report that topological sort is complete"
  else
    "Report cycle in remaining constraints and elements"
  end
end
```

Implement

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

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

\[
\begin{array}{c|ccc}
\text{predecessor_count} & 3 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{c|l}
\text{successors} & 4 \quad 3 \quad 1 \quad 0 \\
\end{array}
\]

targets := successors \([next]\)
from targets, start until targets, after
loop
  freed := targets, item
  predecessor_count \([\text{freed}]\) := predecessor_count
  \([\text{freed}]\) := predecessor_count
  targets, forth
end
```
Implement “Remove from constraints all pairs \([\text{next}, \text{y}]\)” as a loop over the successors of \(\text{next}\):

```plaintext
targets := successors[\text{next}]
from
targets.start until
targets.after
loop
freed := targets.item
predecessor_count[\text{freed}] := predecessor_count[\text{freed}] - 1
end
```

```plaintext
loopy
```

```plaintext
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..."

Finding a candidate (2): on the spot

Complement

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

by

\[
\text{if predecessor_count}[\text{freed}] = 0 \text{ then}
\]

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

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

end

Data structure 4: candidates

candidates: STACK[INTEGER]

-- Items with no predecessor

Instead of a stack, \( \text{candidates} \) 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
Finding a candidate (2)

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:

\(\text{next} := \text{candidates.item}\)

Finding a candidate (3)

Implement the test

"Every member 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 \((n \text{ times})\)

- 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
Detecting cycles

process is 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
if "No more elements" then
    "Report that topological sort is complete"
else
    "Report cycle in remaining constraints and elements"
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:

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