Lecture 20: Topological Sort Algorithm

Overall structure (original)

Given:
- A type \( G \)
- A set of elements of type \( G \)
- A relation \( \text{constraints} \) on these elements

Required:
- An enumeration of the elements in an order compatible with \( \text{constraints} \)

```
class TOPLOGICAL_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
```

Non-uniqueness

In general there are several possible solutions

In practice topological sort uses an optimization criterion to choose between possible solutions.

Overall structure (improved)

```
class TOPLOGICAL_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
```

A partial order is acyclic

The \( \preceq \) relation:
- Must be a partial order: no cycle in the transitive closure of \( \text{constraints} \)
- This means there is no circular chain of the form
  \[ e_0 \preceq e_1 \preceq \cdots \preceq e_n \preceq e_0 \]

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

\[ \{ \text{Dishes, Out}, \text{Museum, Lunch}, \text{Medicine, Lunch}, \text{Lunch, Dishes} \} \]

The relation of interest is:

\[ \text{acyclic} \iff \text{constraints contains no set of pairs} \{ [f_0, f_1], [f_1, f_2], ..., [f_m, f_0] \} \]

When such a cycle exists, there can be no total order compatible with constraints.

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
  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:

```
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
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
```

Intro. to Programming, lecture 20: Topological sort algorithm

This assumes there are no cycles in the input.
The basic algorithm idea

The basic loop scheme

The loop invariant

Terminology

Algorithm scheme

Implementing the algorithm
Intro. to Programming, lecture 20: Topological sort algorithm

Data structures 1: original

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

- **elements**
  - a
  - b
  - c
  - d
  - \(n\) elements

- **constraints**
  - a
  - b
  - a
  - d
  - b
  - d
  - c
  - d
  - \(m\) constraints

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

Basic operations

```
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"
  next := "A member of elements with no predecessor"
  sorted.extend(next)
  "Remove from constraints all pairs of the form \([next, y]\)"
  end
  "Remove 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

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

- **elements**
  - a
  - b
  - c
  - d
  - \(n\) elements

- **constraints**
  - a
  - b
  - a
  - d
  - b
  - d
  - c
  - d
  - \(m\) constraints

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(...), 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
```
Data structure 1: representing elements

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

Data structure 2: representing constraints

- **successors**: ARRAY\[LINKED\_LIST\[INTEGER\]]
  - Items that must appear after any given one

Data structure 3: representing constraints

- **predecessor\_count**: ARRAY\[INTEGER\]
  - Number of items that must appear before a given one

Reminder: basic algorithm idea

Finding a "candidate" (element with no predecessor)

Implement `next := "A member of elements with no predecessors"` as:

- Let `next` be an integer, not yet processed, such that `predecessor\_count[next] = 0`

Finding a candidate (1)

Seems to require an \(O(n)\) search through all indexes: bad!

But wait...
Removing successors

Implement

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

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

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

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

\[
\text{loop}
\]

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

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

\[
[\text{freed}] - 1
\]

\[
\text{targets.forth}
\]

end

end

Removing successors

\[
\begin{array}{c|c|c|c|c}
\text{predecessor_count} & 3 & 1 & 0 & 0 \\
\hline
\text{successors} & x & 2 & 4 & 4 \\
\end{array}
\]

Removing successors

\[
\begin{array}{c|c|c|c|c}
\text{predecessor_count} & 3 & 1 & 0 & 0 \\
\hline
\text{successors} & x & 2 & 4 & 4 \\
\end{array}
\]
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 \( \text{preddecessor\_count}[\text{next}] = 0 \)

We said:

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

Finding a candidate (2): on the spot

Complement

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

by

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

Finding a candidate (2)

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

Finding a candidate (3)

Implement the test

\[
\text{Every member of has a predecessor“}
\]
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 (\( n \) 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
      do
        every member of elements has a predecessor
        next := a member of elements with no predecessor
        sorted, extend(next)
        remove from elements
        remove from constraints all pairs [next, y]
      end
    loop
  end

  if "no more elements" then
    report that topological sort is complete
  else
    report cycle in remaining constraints and elements
  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 [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 heuristic

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.
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