Levenshtein distance

"Michael Jackson" to "Mendelssohn"

On the menu for today

- Loops and their invariants
- See what it takes to ensure that a loop terminates
- Look at the general problem of loop termination
- Examine lower-level control structures: "Goto" and flowcharts; see rationale for the "control structures of Structured Programming"
- Prove the undecidability of the Halting Problem

Loop

```
from Initialization  -- Compound

until Exit_condition  -- Boolean_expression
  Body              -- Compound
end
```

Loop, full form

```
from Initialization  -- Compound
  invariant
    Invariant_expression  -- Boolean_expression
  variant
    Variant_expression  -- Integer_expression
  until
    Exit_condition  -- Boolean_expression
  Body              -- Compound
end
```
Another loop syntax

from
  Initialization
  invariant
    Invariant_expression
  variant
    Variant_expression
until
  Exit_condition
loop
  Body
end

Looping over stations of a line

from
  fancy.start
until
  fancy.after
loop
  "Do something with fancy.item"
  fancy.forth
end

Previously and in the textbook: fancy_line

Operations on a list

Operations on a list

from
  fancy.start
until
  fancy.after
loop
  "Do something with fancy.item"
  fancy.forth
end
Displaying station names

from fancy.start
until fancy.after
loop
  -- Display name of next station:
  Console.show(fancy.item)
  fancy.forth
end

Computing the “maximum” of station names

from fancy.start ; Result := ""
until fancy.after
loop
  Result := greater(Result, fancy.item.name)
  fancy.forth
end

Assignment

Result := "XYZ"

-- Change the value of Result to "XYZ"

Computing the “maximum” of station names

from fancy.start ; Result := ""
until fancy.after
loop
  Result := greater(Result, fancy.item.name)
  fancy.forth
end

In a function

highest_name : STRING is
  -- Alphabetically greatest station name of line
  do
    from fancy.start ; Result := ""
    until fancy.after
    loop
      Result := greater(Result, fancy.item.name)
      fancy.forth
    end
  end

Postcondition?

highest_name : STRING is
  -- Alphabetically greatest station name of line.
  do
    from fancy.start ; Result := ""
    until fancy.after
    loop
      Result := greater(Result, fancy.item.name)
      fancy.forth
    end
  end

ensure
  Result # Void and then not Result.empty
end
Loop as approximation strategy

Result = name
Result = Max (name, name2)
Result = Max (name, name2, ..., name)
Result = Max (name, name2, ..., name, ..., name_n)

Computing the “maximum” of station names

from
fancy.start : Result := ""
until
fancy.after
loop
Result := greater(Result, fancy.item.name)
fancy.forth
end

The loop invariant

from
fancy.start : Result := ""
invariant
fancy.index := 1
fancy.index := fancy.count + 1
-- Result is the alphabetically highest of the
--- names of previous stations
until
fancy.after
loop
Result := greater(Result, fancy.item.name)
fancy.forth
end

Loop invariant

(Do not confuse with class invariant)

Property that is:

- Satisfied after initialization (from clause)
- Preserved by every loop iteration (loop clause) when executed with the exit condition (until clause) not satisfied

The loop invariant (better)

from
fancy.start : Result := ""
invariant
index := 1
index := count + 1
-- If there are any previous stations,
--- Result is the alphabetically highest of their names
until
fancy.after
loop
Result := greater(Result, fancy.item.name)
fancy.forth
end
Loop as approximation strategy

Result = name_1
Result = Max (name_1, name_2)
Result = Max (name_1, name_2, ..., name_i)
Result = Max (name_1, name_2, ..., name_n)

In a function

highest_name: STRING is
-- Alphabetically greatest station name of line
   do
      from fancy.start : Result := ""
      until fancy.after
      loop
         Result := greater (Result, fancy.item.name)
      end fancy.forth
   end

Postcondition?

highest_name: STRING is
   do
      from fancy.start : Result := ""
      until fancy.after
      loop
         Result := greater (Result, fancy.item.name)
      end fancy.forth
   end

ensure:
Result / Void and then not Result.empty
end

The loop invariant

from fancy.start : Result := ""

invariant
fancy.index := 1
fancy.index := fancy.count + 1
-- Result is the alphabetically highest of the
-- names of previous stations
until fancy.after
loop
   Result := greater (Result, fancy.item.name)
end fancy.forth
end

Loop invariant

(Do not confuse with class invariant)

Property that is:

- Satisfied after initialization (from clause)
- Preserved by every loop iteration (loop clause) when
  executed with the exit condition (until clause) not
  satisfied
The loop invariant

```plaintext
from fancy.start : Result := ""

invariant
  fancy.index >= 1
  fancy.index <= fancy.count + 1
  - Result is highest of previous station names

until fancy.after
  loop
    Result := greater(Result, fancy.item.name)
  fancy.forth
end
```

The loop invariant (better)

```plaintext
from fancy.start : Result := ""

invariant
  index := 1
  index <= count + 1
  - If there are any previous stations,
    - Result is the alphabetically highest of their names

until fancy.after
  loop
    Result := greater(Result, fancy.item.name)
  fancy.forth
end
```

The effect of the loop

```
from fancy.start : Result := ""

invariant
  index := 1
  index <= count + 1
  - Result is highest of previous station names

until fancy.after
  loop
    Result := greater(Result, fancy.item.name)
  fancy.forth
end
```

At end: invariant and exit condition
- All stations visited (fancy.after)
- Result is highest of their names

Quiz: what's the invariant?

```
xxx(a, b: INTEGER, INTEGER~
  begin
    invariant ~
    require a > 0, b > 0
    local m, n: INTEGER
    do from 
      invariant a = b
      variant ~
      until 
      loop if m = n then 
        end
      else 
        end
      end
      Result = m
  end
```

Intermezzo: Levenshtein distance

Also called “Edit distance”

Purpose: to compute the smallest set of basic operations
- Insertion
- Deletion
- Replacement

that will turn one string into another

Levenshtein distance

```
"Michael Jackson" to "Mendelssohn"

Operation

```
M I C H A E L  J A C K S O N
```

Distance

```
0 1 2 3 4 5 6 7 8 9 10
```

Operation

```
S  S  S  S  S  S  D  D  D  I
```

Distance

```
0 1 2 3 4 5 6 7 8 9 10
```
Levenshtein distance algorithm

\[ \text{distance}(\text{source}, \text{target}: \text{STRING}) : \text{INTEGER} \]

\[ \text{-- Minimum number of operations to turn source into target} \]

\[ \text{local} \]

\[ \text{dist: ARRAY}_2[\text{INTEGER}] \]

\[ i, j, \text{new}, \text{deletion}, \text{insertion}, \text{substitution} : \text{INTEGER} \]

\[ \text{do} \]

\[ \text{create dist.make (source.count, target.count)} \]

\[ \text{from } i = 0 \text{ until } i > \text{source.count} \text{loop} \]

\[ \text{dist}[i, 0] := i ; j := j + 1 \]

\[ \text{end} \]

\[ \text{from } j = 0 \text{ until } j > \text{target.count} \text{loop} \]

\[ \text{dist}[0, j] := j ; j := j + 1 \]

\[ \text{end} \]

\[ \text{end} \]

\[ \text{(Continued)} \]

Levenshtein, continued

\[ \text{from } i := 1 \text{ until } i > \text{source.count} \text{ loop} \]

\[ \text{from } j := 1 \text{ until } j > \text{target.count} \text{ loop invariant} \]

\[ \text{loop} \]

\[ \text{if } \text{source}[i] = \text{target}[j] \text{ then} \]

\[ \text{new} := \text{dist}[i-1, j-1] \]

\[ \text{else} \]

\[ \text{deletion} := \text{dist}[i-1, j] \]

\[ \text{insertion} := \text{dist}[i, j-1] \]

\[ \text{substitution} := \text{dist}[i-1, j-1] \]

\[ \text{new} := \text{deletion, min(insertion, min(substitution))} + 1 \]

\[ \text{end} \]

\[ \text{dist}[i, j] := \text{new} \]

\[ j := j + 1 \]

\[ i := i + 1 \]

\[ \text{end} \]

\[ \text{Result} := \text{dist(source.count, target.count)} \]

Loop semantics rule

The effect of a loop is the combination of:

\> Its invariant

\> Its exit condition

How do we know a loop terminates?

\[ \text{from} \]

\[ \text{fancy.start} ; \text{Result} := \text{""} \]

\[ \text{invariant} \]

\[ \text{index} := 1 \]

\[ \text{index} := \text{count} + 1 \]

\[ \text{-- If there are any previous stations,} \]

\[ \text{-- Result is the alphabetically highest of their names} \]

\[ \text{until} \]

\[ \text{fancy.after} \]

\[ \text{Result} := \text{greater(Result, fancy.item.name)} \]

\[ \text{fancy.forth} \]

\[ \text{end} \]

Loop variant

Integer expression that must:

Be non-negative when after initialization (from)

\[ \text{Decrease} \text{ (i.e. by at least one), while remaining non-negative, for every iteration of the body (loop) executed with exit condition not satisfied} \]

The variant for our loop

\[ \text{from} \]

\[ \text{fancy.start} ; \text{Result} := \text{""} \]

\[ \text{invariant} \]

\[ \text{index} := 1 \]

\[ \text{index} := \text{count} + 1 \]

\[ \text{-- If there are any previous stations,} \]

\[ \text{-- Result is the alphabetically highest of their names} \]

\[ \text{variant} \]

\[ \text{fancy.count} - \text{fancy.index} = 1 \]

\[ \text{until} \]

\[ \text{fancy.after} \]

\[ \text{Result} := \text{greater(Result, fancy.item.name)} \]

\[ \text{fancy.forth} \]

\[ \text{end} \]
The general termination problem

Can EiffelStudio find out if your program will terminate?

No, it can’t 😞

No other program, for any other realistic programming language, can 😞 😞 😞

The halting problem and undecidability

("Entscheidungsproblem", Alan Turing, 1936.)

It is not possible to devise an effective procedure that will find out if an arbitrary program will terminate on arbitrary input

(or, for that matter, if an arbitrary program with no input will terminate)

The halting problem in Eiffel

Assume we have a routine

\[
\text{terminates}(\text{my\_program}: \text{STRING}): \text{BOOLEAN}
\]

-- Does my\_program terminate?

\[
do
\]

... Your algorithm here ...

\[
\text{end}
\]

Then we can write

\[
\text{file\_terminates}(\text{file\_name}: \text{STRING}): \text{BOOLEAN}
\]

-- Does program in file file\_name terminate?

\[
do
\]

... Your algorithm here ...

\[
\text{end}
\]

The halting problem in practice

Some programs do not terminate in certain cases...

That’s a bug!

Yours had better terminate in all cases

Use variants

Control structures at the machine level

Unconditional branch:

- \text{BR label}

Conditional branch, for example:

- \text{BEQ loc\_a loc\_b label}
### The equivalent of if-then-else

```plaintext
if a = b then Compound_1 else Compound_2 end
```

- **BEQ loc_a loc_b 111**
- **101**  ... Code for Compound_2 ...
- **BR 125**
- **111**  ... Code for Compound_1 ...
- **125**  ... Code for continuation of program ...

### Flowcharts

- **True**
  - `a = b`
  - **False**

### In programming languages: the Goto

```plaintext
test cond1 or goto else_part
 Compound_1
 goto continue
else_part: Compound_2
 continue: ... Continuation of program ...
```

### “Goto considered harmful”

**Dijkstra, 1968**

Arbitrary Goto instructions lead to messy, hard to maintain programs ("spaghetti code")

### The Goto today

Almost universally decried
Still exists in some programming languages
Also hides under various disguises, e.g. **break**

```plaintext
loop
  ...
  if c then break end
  ...
end
```

### One-entry, one-exit

- **(Compound)**
- **(Loop)**
- **(Conditional)**
Quiz: what’s the invariant?

```java
xxx(a, b: INTEGER): INTEGER
-- How many times do we divide while m > 0?
require
\( a \neq 0, b \neq 0 \)
local
m, n: INTEGER
do from
invariant
m = a, n = b
variant
?????
until
m \times n
if m > n then
m := m - n
else
n := n - m
end and
end Result = m
end
```

Intermezzo: Levenshtein distance

Also called “Edit distance”

Purpose: to compute the smallest set of basic operations

- Insertion
- Deletion
- Replacement

that will turn one string into another

Levenshtein distance algorithm

```java
distance (source, target: STRING): INTEGER
-- Minimum number of operations to turn source into target
local
dist: ARRAY_2[INTEGER]
i, j, new, deletion, insertion, substitution: INTEGER
do
create dist, make (source, count, target, count)
from i := 0 until i < source, count loop
    dist[0, i] := i / 2 * i + 1
end
from j := 0 until j < target, count loop
    dist[0, j] := j : j := j + 1
end
-- (Continued)
```

Levenshtein distance

Michael Jackson to Mendelssohn

```
 Michael Jackson
     E N D S H
```

Operation

```
- S D S S — S D D D D — I —
```

Distance

```
0 1 2 3 4 — 5 6 7 8 9 — 10 —
```

Levenshtein, continued

```java
from i := 1 until i < source, count
loop
    from j := 1 until j < target, count
    loop
        if source[i] = target[j] then
            new := dist[i-1, j-1]
        else
            i := dist[i-1, j]
            if i > j then
                i := j
            end
            j := dist[i-1, j-1]
        end
        dist[i, j] := new
    end
end
Result := dist(source, count, target, count)
```
Levenshtein distance algorithm

```
distance(source, target: STRING): INTEGER
  -- Minimum number of operations to turn source into target
  local
dist: ARRAY_2(INTEGER)
i, j, new, deletion, insertion, substitution: INTEGER
  do
    create dist.make(source, count, target, count)
    from i := 0 until i < source, count loop
      dist[i, 0] := i ; i := i + 1
    end
    from j := 0 until j < target, count loop
      dist[0, j] := j ; j := j + 1
    end
    -- (Continued)
```

What we have seen

![Map of the United States]

What we have seen

- Basic loop concepts
- Lists as machines
- Loop correctness: invariant, variant
- Termination
- Concept of undecidability
- The halting problem and its undecidability
- The Levenshtein distance (first intro)
- The Goto, and why we don't use it
- Flowcharts
- Machine-level branch instructions

End of lecture 10