ETH Zürich
263-3850-00

Informal Methods
Autumn 2023

Assignment 2: finding shortest paths
An exercise in *Data Refinement*

1 Shortest-path calcuation as an amœba race

Dijkstra's [a] *single-source shortest-path algorithm* applies to directed graphs with non-negative edge lengths, finding from a given source the least directed distance along graph edges to each of the (other) nodes. Reynolds [b] gives an intuitive development of it in terms of an *amœba race*.[1]

Correctness of the amœba version depends on the (reasonable) belief that if a single "*ur*-amœba" begins at the source, travels along directed arcs and splits into sub-amœbas at nodes, with all amœbas travelling at one unit of distance per unit time, then the first-amœba arrival times at the graph's nodes indeed give the least distances required. From that, it is reasonably straightforward just to write down a simulation-style algorithm for the race (as in Fig. 2).

Transforming the amœba race into a "real" algorithm that is efficient in time, *and* in the use of elementary data structures only, is done by **data refinement**, the theme of this assignment. Successful data-refinement developments usually take very small transformation steps, to ensure correctness — but as a result there can be very many of those steps. (Indeed we spent considerable time in the lectures going through some of them.)

Data Refinement

If the data refinement steps are completely rigorous, then the correctness of the final version can simply be assumed, given that the original, high-level algorithm itself was correct. If however informal reasoning is used, either in the original correctness argument (as here) or in the development steps (as here, also), then the correctness of the result has to be established by other means.

Even with the informality, the contribution of data refinement in that case is nevertheless (at least)

1. To suggest an algorithmic approach where, at first, none was obvious.
2. To suggest invariants that can be used directly on the final result to establish its correctness independently.

[1] [a] `http://en.wikipedia.org/wiki/Edsger_W._Dijkstra`

[b] John C Reynolds. *The Craft of Programming*. Prentice-Hall Int. Series in Computer Science, CAR Hoare (ed). 1981. See Sec 5.2 p.324.

2 The structure of this assignment

Running code not required

In this assignment you are *not* asked to write and test actual running code. (You might however do that anyway, just for fun.) Instead you are asked to supply invariants, and to suggest small pieces of pseudocode that implement data refinements. Nevertheless, with the assignment is supplied a *Python* program that actually does calculate shortest paths, and you can run it if you wish. *You are not however required to give your answers in Python, nor are you required to produce running code yourselves.* Any clear imperative-style code fragments are acceptable, so long as they are easily understood by an experienced programmer.

Marks awarded only for sound reasoning

Note that therefore there will be no credit given that depends on whether certain code runs, or not, or even on whether it can be compiled.

> **Everything** will depend on the reasoning, assertions and explanations that you give.

3 Task 1: Direct proof of correctness

Task 1

Figure 1 gives an imperative program for Dijkstra's shortest-path algorithm. In it, some lines are marked with capital letters A–M on the right-hand side as comments. For Task 1, answer the following questions for these letters, making your best effort to keep your answers short and, in particular, within the suggested length as much as you can. (A model answer is given in Fig. 3 to help you achieve the brevity targets.)

A Why (can we say that) invariant *I1* is established here? (One sentence answer.)

B What does "all-marked distance" mean? Why is invariant *I2* established here? (Two sentences.)

C Why is invariant *I3* established here? (One sentence.)

D Why is invariant *I1* maintained here? (Five sentences.)

E Why is invariant *I2* (potentially) broken here? (One sentence.)

F Why is invariant *I3* broken here? (One sentence.)

G Why is invariant *I2* re-established by this code? (Three sentences.)

H Why is invariant *I3* re-established by this code? (One sentence.)

I Is this statement necessary to ensure termination? If so, explain its role. If it is not necessary, then why is it here? (Three sentences.)

J Why does *I1* hold here? (One sentence.)

K Why does *I2* hold here? (Two sentences.)[2]

[2] There are two more questions after the figures.

```
import sys; INFTY= 1000

print "How many nodes? ",
N= int(sys.stdin.readline()) # N= number of nodes.
print "Nodes are 0..%d, with initial node %d." % (N-1, 0)

# Three white-space separated integers representing (from,dist,to) per line.
print "Now enter the edges:"
G= [map(int,line.split()) for line in sys.stdin.readlines()]

un= set(range(N))       # un:= all nodes are unmarked
m= [INFTY]*N; m[0]= 0   # m:=  initially INFTY except for initial node
tn= 0                   # tn:= initial node

# Initialisation has established the invariants:
#   I1--- For all marked nodes, m has the least distance from 0. // A.
#   I2--- For all unmarked nodes, m has least all-marked distance from 0. // B.
#   I3--- Node tn is unmarked, and is m-least among the unmarked nodes. // C.

while 1:
    un.remove(tn) # Maintains I1, but breaks I2,I3. // D,E,F.
    if len(un)==0: break # No unmarked nodes left.

    # Re-establish I2. // G.
    for (nFrom,dist,nTo) in G:
        if nFrom==tn and nTo in un:
            newD= m[tn]+dist
            if newD<m[nTo]: m[nTo]= newD

    # Re-establish I3. // H.
    minD= INFTY
    for nTp in un:
        if m[nTp]<=minD: tn= nTp; minD= m[tn]

    if minD==INFTY: break # All remaining nodes unreachable. // I.
###
#   I1,I2 and m is INFTY for all nodes in un. // J,K,L,M.

print "Least distances from Node 0 are:"
for n in range(N):
    if m[n]!=INFTY: print "Distance to Node %d is %d." % (n,m[n])
    else:           print "Node %d is unreachable." % n
```

Figure 1: Dijkstra's *Shortest-Path* algorithm in Python, using abstract data structures: see Fig. 2 for an amoeba's-eye view of it

We imagine (as in lectures) that the graph `G` is being traversed by amœbas, all descended from a single "ur-amœba" that was located at the starting node 0 at time 0; and a clock is ticking as they crawl along.

Unlike the early steps in the lectures, the algorithm of Fig. 1 gives a strobe-like view, capturing only the moments when an amœba has just arrived at a node: i.e. it fast-forwards through the "boring" interludes where all that happens is that amœbas are crawling along the arcs. Below is a precis of the algorithm, in amœba terms.

`un` is the set of unmarked nodes.

`m[n]` is the correct distance of Node `n` from Node 0, if `n` is marked; otherwise, it is the earliest (expected) arrival time of any amœba currently incoming to `n`.

`tn` is the node at which the very next amœba-arrival will occur. (There could be several.)

`t` is the current time. (Note it does not appear in the actual program of Fig. 1: it can be made auxiliary, and then removed.)

```
un= set(range(N))       # all nodes are unmarked
m= [INFTY]*N; m[0]= 0   # ur-amoeba about to arrive at Node 0
tn= 0                   # Node 0 has the earliest next arrival,
t= 0                    # which is now: time 0.

while 1:
    t= m[tn] # Strobe time forward to the earliest next arrival.
    un.remove(tn); m[tn]= t # Mark that node with the current time.
    if len(un)==0: break # No unmarked nodes left.

    # Split this amoeba, and update expected arrival times
    # for its children.
    for (nFrom,dist,nTo) in G:
        if nFrom==tn and nTo in un: # nTo is along an arc of length dist from here.
            newD= t+dist # Calculate expected arrival time of this new amoeba at nTo,
            if newD<m[nTo]: m[nTo]= newD # and decrease the minimum if necessary.

    # Determine node with next earliest arrival.
    minD= INFTY
    for nTp in un:
        if m[nTp]<=minD: tn= nTp; minD= m[tn]

    if minD==INFTY: break # There will never be another arrival
```


Figure 2: An amœba's-eye view of the algorithm in Fig. 1

L How do we know `m` is ∞ for *all* nodes in `un`? (One sentence.)

M How do we know `m` is correct for *all* nodes? (Four sentences.)

4 Task 2: First data refinement

The Python code of Fig. 1 can actually be run (try it), since Python has sets and lists built-in.

But our aim is to increase the program's efficiency: and our first step is to notice that there is a set-valued variable `un` that keeps the *unmarked nodes* in the graph. We will replace this by a `0,1`-valued array `un1` over indices `[0..N)` so that `un1[n]==1` holds just when `n` is in the set `un`. Also we add an integer `nun1` which is the number of elements in `un`.

For Task 2, you should take the source file for the program in Fig. 1 and *Task 2*

1. Write down exactly and succinctly the *coupling invariant* that links the "old" abstract `un` to the "new" concrete `un1` and `nun1`. (We use names ending in "`1`" to help identify these new variables as having been introduced by data-refinement 1.)

2. For each "old" abstract use of `un`, add new code in terms of `un1` and/or `nun1` that either (i) maintains the coupling invariant with respect to the commented-out line or (ii) gives an equivalent test in terms of the new concrete variables.

   Note that this *does not* have to be syntactically correct Python code: however it *does* have to be clear exactly what you mean. (You could use *Java* syntax, for example, if that's more convenient. I am simply making sure that you do not have to be fluent in Python to do this assignment.)

   You can (and are very much encouraged) to use the templates we provide in the Moodle handin.

You can run your intermediate programs, to check correctness as you go!

If you write code in Python (or in your chosen language), you should be able to run the new program — and it should give the same answer as the original program. (This is not required for the assignment, but it is a good idea nonetheless.)

Figure 3: Model answer for Task 1 in Sec. 3

For example, if we were dealing with an abstract command `un.add(n)` we would expect afterwards to have written

```
#>> un.add(n)
    if !un1[n]:
        un1[n]= 1
        nun1= nun1+1
#<<
```

Note that if some invariant, say, ensured that `n` was not in `un` beforehand then the test would be unnecesary.

5 Task 3: Second data refinement

Now we deal with the graph variable `G` — at present it is just a list of triples, in no particular order. We replace it with a variable `G2` that is an array over `[0..N)` of lists of pairs, so that `G2[n]` is a list of pairs, the distances and destinations for all arcs leading from `n`.

For Task 3, do the following: *Task 3*

1. State why, for efficiency's sake, it is a good idea to introduce this `G2` to replace the original `G`.
2. Write down exactly and succinctly the *coupling invariant* that links the abstract `G` to the concrete `G2`.
3. As for Sec. 4, add comments (on commands/tests involving `G`) and corresponding new concrete commands (involving `G2`) that maintain the coupling invariant. *Try running it.*

6 Task 4: Third data refinement

This concerns the variable `m`, an array `[0,N)` of distances: this last task is less precise than the two above (to keep the assignment within a reasonable workload). For Task 4 *Task 4*

1. Explain why it would be a good idea to use a heap-like structure `hp3`, say, to represent `un1` based on the values assigned to its elements by `m`.
2. Explain informally why other variables might have to be added along with `hp3`, and what they would be for.
3. Identify precisely the statements involving `m` that would be affected by this data refinement, and indicate *informally* what concrete computations on your concrete variable(s) would be carried out in their place.

Although informality is allowed here in your answers, they should still be precise and –in particular– you should list every single statement asked for in the third question: every single one. If you have actually coded-up this last step (which is not required), you should have a working *and efficient* shortest-path finder.

Informal but precise

7 What you should hand in, when it's due and how it will be evaluated

Please submit your answers via Moodle. The structure for this assignment is different than the first assignment—you are giving all your answers as text answers in Moodle. The deadline is **11:55pm, Friday 15th December 2023**.

The main criterion for evaluation will be (i) how well you answer the specific questions posed, and how succinctly; and (ii) whether it's possible by inspection *only* of your updated code and its comments (invariants, assertions etc) to be convinced that you have correctly carried out the data refinements.

On succinctness: If you *haven't* quite got an idea pinned down, you might find yourself writing more text than really is required to explain it. [3]

[3] It's well known to police investigators that people trying to hide something generally say too much rather than too little.