COMP 354: INTRODUCTION TO SOFTWARE ENGINEERING

PROJECT MANAGEMENT APPLICATION
TEST DESIGN DOCUMENTATION

Team B

Alex Huot

Dmitri Novikov

David Gurnsey

Jun Hui Chen

Philip Eloy

Robert Wolfstein

Nicholas Francoeur

Mehdi Jamal Mokhtar

Nivine Shebarou

Lee Kelly

Steven Di Lullo

Tuesday, August 26th, 2014

Revision History:

Date Created: Aug 17th, 2014

Most Recent Update: Aug 26th, 2014

Previous Updates:

Revision Date	Contributor	Summary
Aug 26th, 2014	Alex Huot	Created doc based on passed iteration template.

Table of Contents

Index of Tables

Document Description:

Introduction:

This document provides design documentation for Team B's third iteration deliverable. The chosen functions to test, justification for said choices, and all test-design is provided here. The test implementations are included in a separate package in the final source code submission.

A note on selected functions:

The requirements for this iteration's testing were to select 5 functions suitable for both white-box and black-box testing. Unfortunately, without important changes to the logic of our code, we had no such functions to test. In fact, several of our black-box functions are segments of code originally part of larger algorithms, which we extracted in the form of helper functions ideal for this type of test. Despite this, those tests suitable for black-box had very simple decision logic, and those suitable for white-box seldom took more than 1 parameter with a specifiable range. For this reason, we opted to perform white-box and black-box tests on a different set of 5 functions for each. We felt this would better demonstrate our understanding of these testing principles, and offer more interesting results for discussion.

1.0 Choice of Functions, Justification and Input Ranges

This section outlines the choice of our 5 black-box and 5 white-box functions. Here we also provide a short description of each function and the justification for our choices. The input ranges are specified for each.

2.0 White Box Testing

The Control Flow Graphs (CFGs), cyclomatic-complexity and **basis-path coverage** calculations for our white-box functions are provided here.

3.0 Black Box Testing

This section provides the wort-case boundary testing design for our black-box tests. As we are not performing any white-box tests for these functions, we opted to omit the CFGs and basis-path coverage calculations for these functions. The black-box tests were performed using the **worst-case boundary value-analysis** method.

4.0 Test Headers

All test headers provided here, specifying the conditions, input and outputs for each test case.

5.0 Discussion

The concluding discussion of insights gained from both test-design as well as implementation.

1.0 Choice of Functions, Justification and Input Ranges

1.1 Choice of White-Box Functions

The 5 selected functions for our white-box tests are:

1. Class: GanttNetwork

   ```
   public void backwardPass(Activity activity)
   ```

Description:

This function set's the latest start and finish values for an activity when performing the backward pass of the Gantt Chart generation.

Justification:

This function has complex decision logic, relying on the start and finish values of potentially several dependents. This will be made more apparent in section 3.0, where the cyclomatic-complexity and CFGs are displayed.

Input Ranges:

This function takes as input an `Activity`, which is a complex data type. The white-box test shall be performed by providing an `Activities` with specific attributes and dependents to ensure all paths of the basis-path coverage calculation are travelled.

2. Class: PertNetwork

```
private void forwardPass()
```

Description:

This function performs the forward pass in the Pert Network calculation. It sets the expected date and standard deviation.

Justification:

This function also includes a complex series of decisions to be taken, with several if-statements and for-loops. It is an ideal candidate for white-box test design.

Input Ranges:

This function takes no inputs, and uses only the local variables of the `PertNetwork` class to perform its calculations. A `PertNetwork` object will be created for the test with instance variables set such that each basis-coverage path is travelled.

3. Class: PertNetwork

```
public static getArrowNetwork(ArrayList<Activity> activities)
```

Description:

This function returns the arrow network based on a given set of activities.

Justification:

Once again, a series of conditions placed on these activities increases the cyclomatic-complexity for this function.

Input Ranges:

The resulting arrow network is dependent on the list of input activities. The number of activities and their relationships (in the form of dependents and precedents) will be manipulated to cover the required paths.

4. Class: EarnedValue

```
public void calculateActualCost()
```

Description:

Calculates the actual cost for a `Project` based on all it's activities and their percentage of completion.

Justification:

The dependence on an `Activity` list are what make this function structurally ideal for white-box testing. Several potential paths, perfect for white-box.

Input Ranges:

Takes no parameters; all calculations are performed on the `EarnedValue` object's `Project` and its `Activities`

5. Class: `MilestoneNode`

```
public String toStringArrows(String type)
```

Description:

This function generates a string describing the `MilestoneNode`'s related activities, namely those to complete by the milestone, or those available after the milestone.

Justification:

As with previous white-box tests, the decision logic entailed by this function make it an interesting white-box test.

Input Ranges:

The function takes a type as input, and depending on this type and the `MilestoneNode`'s local `Activity` list, a descriptive string will be generated.

1.2 Choice of Black-Box Functions

The 5 selected functions for our black-box tests are:

1. Class: EarnedValue

```
public static double getActivityScheduleValue(double activityEarlyFinish,
      double activityLateFinish, double activityPlannedVal,
      double activityDuration, double daysSinceStart)
```

Description:

This returns the scheduled value for an `Activity`.

Justification:

By taking several numerical inputs with specifiable range, this function is an ideal candidate for worst-case boundary value analysis.

Input Ranges:

```
double activityEarlyFinish = [0, 731] // Days. Max set to 2 years
double activityLateFinish  = [0, 731] // Days. Max set to 2 years
double activityPlannedVal  = [0, 1000000] // $. Max set to 1 million
double activityDuration    = [0, 731] // Days. Max set to 2 years
double daysSinceStart      = [0, 73] // Days. Max set to 2 years less a day
```

2. Class: Project

```
public static boolean areValidPercentages(double percentage1,
      double percentage2)
```

Description:

Verifies if percentages are valid.

Justification:

To variables with bounded inputs, ideal for boundary testing.

Input Ranges:

```
double percentage1 = [0, 1] // Percentage in decimal format
double percentage2 = [0, 1] // Percentage in decimal format
```

3. Class: Project

```
public static boolean areValidValues(double budgetAtCompletion,
      double actualCost, double earnedValue)
```

Description:

Verifies if valid values are fed to `Project's` constructor.

Justification:

This function performs an important check on the bounds of the specified values. Ideal for black-box testing.

Input Ranges:

```
double budgetAtCompletion = [0, 10 000 000] // $. Max fixed to 10 million
double actualCost         = [0, 10 000 000] // $. Max fixed to 10 million
double earnedValue        = [0, 10 000 000] // $. Max fixed to 10 million
```

4. Class: Activity

```
public static boolean areValidValues(double mostLikely, double optimistic,
      double pessimistic)
```

Description:

Verifies that the time values passed to `Activity` constructor are valid.

Justification:

These are bounded values, making them ideal for black-box testing

Input Ranges:

```
double mostLikely  = [0, 731] // Days. Max set to 2 years
double optimistic  = [0, 731] // Days. Max set to 2 years
double pessimistic = [0, 731] // Days. Max set to 2 years
```

5. Class: Activity

```
public static boolean areValidPercentAndCost(double percentComplete,
      double actualCost)
```

Description:

Verifies the specified values passed to `Activity` constructor are valid.

Justification:

These values are bounded, making them ideal for black-box testing.

Input Ranges:

```
double percentComplete = [0, 1] // % in decimal format
double actualCost      = [0, 10 000 000] // $. Max set to 10 million
```

2.0 White-Box Testing

For each of the above specified white-box functions, the following information is provided:

- Code
- CFG
- Cyclomatic-complexity
- Basis-path coverage test suite
    - For each test case, the input required to follow this path is provided in bullet form; each bullet represents the necessary input needed to go along the correct path. If there exists a contradictory input, the test case will be labeled **INFEASIBLE**.

2.1 GantNetwork – backwardPass()

```java
// Class:   GanttNetwork
private void backwardPass(Activity activity) {

    double latestStart = Integer.MAX_VALUE, start;
    Activity constrainer = null;

    for(Activity dep : activity.getDependents()){

        start = dep.getLatestStart();

        if(start < latestStart){
            latestStart = start;
            constrainer = dep;
        }

    }
    if(activity.equals(this.getFinish())){
        activity.setLatestFinish(activity.getEarliestFinish());
        activity.setLatestStart(activity.getEarliestStart());
    }
    else{
        activity.setLatestFinish(constrainer.getLatestStart());
        activity.setLatestStart(constrainer.getLatestStart() - activity.getDuration());
    }

    if(activity.equals(this.getStart())){
        return;
    }

    for(Activity pre : activity.getPrecedents()){
        backwardPass(pre);
    }


}

```

Figure 1: backwardPass() - Code

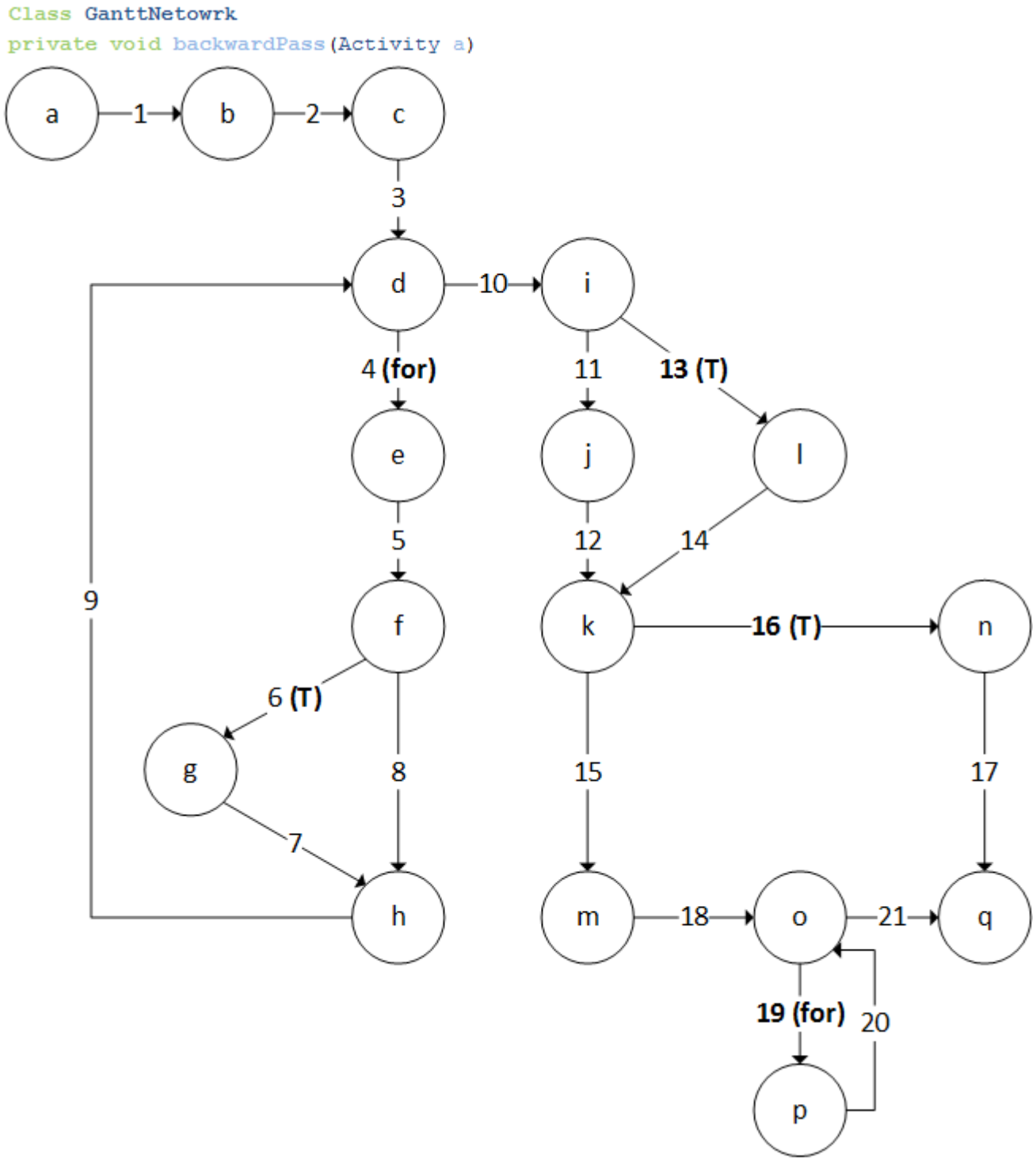

Figure 2: backwardPass() - CFG

Cyclomatic-Complexity:

Complexity = num. Decision nodes + 1

= 5 + 1

= **6**

Basis-Path Coverage:

Test Cases: {

1: <1,2,3,4,5,6,7,9,10,11,12,15,18,19,20,21>

2: <1,2,3,10,11,12,15,18,19,20,21>

3: <1,2,3,4,5,8,9,10,11,12,15,18,19,20,21>

4: <1,2,3,4,5,6,7,9,10,13,14,15,18,19,20,21>

5: <1,2,3,4,5,6,7,9,10,11,12,16,17>

6: <1,2,3,4,5,6,7,9,10,11,12,15,18,21>

}

Test 1 Input:

- Activity(a) with ONE dependent(d)
- d.start < Integer.MAX_VALUE
- Activity is NOT last(end-dummy)
- Activity is NOT start(start-dummy)
- Activity has ONE precedent(p)

Test 2 Input:

- Activity(a) with NO dependents
- a is NOT last(end-dummy)
- a is NOT start(start-dummy)
- a has ONE precedent(p)

Test 3 Input:

- Activity(a) with ONE dependent(d)
- d.start > Interger.MAX_VALUE

INFEASIBLE

Test 4 Input:

- Activity(a) with ONE dependent(d)
- d.start < Integer.MAX_VALUE
- a is last(end-dummy)
- a is NOT start(start-dummy)
- a has ONE precedent(p)

Test 5 Input:

- Activity(a) with ONE dependent(d)
- d.start < Integer.MAX_VALUE
- a is NOT last(end-dummy)
- a is start(start-dummy)

Test 6 Input:

- Activity(a) with ONE dependent(d)
- d.start < Integer.MAX_VALUE
- a is NOT last(end-dummy)
- a is NOT start(start-dummy)
- a has NO precedents

2.2 PertNetwork – forwardPass()

```
1   // Class:   PertNetwork
2   private void forwardPass() {
3
4           double  expectedDate = 0,
5                   expectedSrdDev = 0,
6                   maxStdDev = 0;
7
8           for(MilestoneNode n : this.graph.getNodes().keySet()){
9
10              expectedDate = 0;
11              expectedSrdDev = 0;
12              maxStdDev = 0;
13
14              for(Activity a : n.getInArrows()){
15                  if(a.getEarliestFinish() > expectedDate){
16                      expectedDate = a.getEarliestFinish();
17                  }
18              }
19
20              for(Activity a : n.getInArrows()){
21
22                  expectedSrdDev = Math.pow(a.getDurationStandardDevitation(), 2);
23
24                  for(MilestoneNode n1 : n.getPrecedents()){
25                      if(n1.getOutArrows().contains(a)){
26                          expectedSrdDev += Math.pow(n1.getStandardDeviation(), 2);
27                      }
28                  }
29
30                  if(Math.sqrt(expectedSrdDev) > maxStdDev){
31                      maxStdDev = Math.sqrt(expectedSrdDev);
32                  }
33              }
34
35              n.setExpectedDate(expectedDate);
36              n.setStandardDeviation(maxStdDev);
37          }
38
39      }
```

Figure 3: forwardPass() - Code

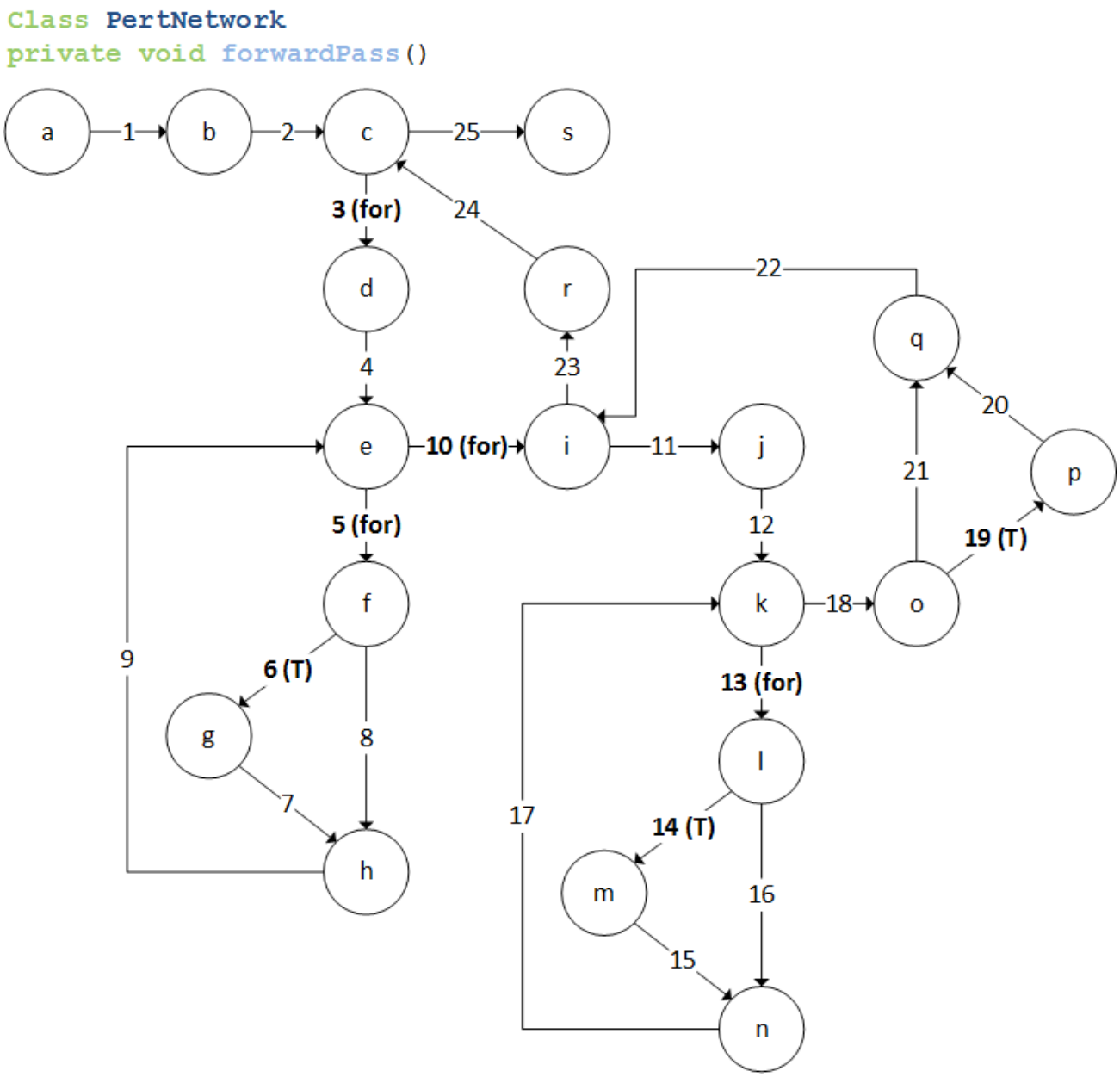

Figure 4: forwardPass() - CFG

Cyclomatic-Complexity:

Complexity = num. Decision nodes + 1

= 6 + 1

= **7**

Basis-Path Coverage:

Test Cases: {

1: <1,2,3,4,5,6,7,9,10,11,12,13,14,15,17,18,19,20,22,23,24,25>

2: <1,2,25>

3: <1,2,3,4,10,11,12,13,14,15,17,18,19,20,22,23,24,25>

4: <1,2,3,4,5,8,9,10,11,12,13,14,15,17,18,19,20,22,23,24,25>

5: <1,2,3,4,5,6,7,9,10,23,24,25>

6: <1,2,3,4,5,6,7,9,10,11,12,18,19,20,22,23,24,25>

7: <1,2,3,4,5,6,7,9,10,11,12,13,16,17,18,19,20,22,23,24,25>

8: <1,2,3,4,5,6,7,9,10,11,12,13,14,15,17,18,21,22,23,24,25>

}

Test 1 Input:

- PertNetwork has ONE MilestoneNode(n)
- n has ONE inArrow(a)
- a.earliestFinish > 0
- n has ONE inArrow(a)
- n has ONE precedent(p)
- p contains a
- sqrt(expectedSrdDev) > 0

Test 2 Input:

- PertNetwork has NO MilestoneNodes

Test 3 Input:

- PertNetwork has ONE MilestoneNode(n)
- n has NO inArrows
- n has ONE inArrow(a)

INFEASIBLE

Test 4 Input:

- PertNetwork has ONE MilestoneNode(n)
- n has ONE inArrow(a)
- a.earliestFinish < 0

INFEASIBLE

Test 5 Input:

- PertNetwork has ONE MilestoneNode(n)
- n has ONE inArrow(a)
- a.earliestFinish > 0
- n has ONE inArrow(a)
- n has NO inArrows

INFEASIBLE

Test 6 Input:

- PertNetwork has ONE MilestoneNode(n)
- n has ONE inArrow(a)
- a.earliestFinish > 0
- n has ONE inArrow(a)
- n has NO precedent
- sqrt(expectedSrdDev) > 0

Test 7 Input:

- PertNetwork has ONE MilestoneNode(n)
- n has ONE inArrow(a)
- a.earliestFinish > 0
- n has ONE inArrow(a)
- n has ONE precedent(p)
- p DOES NOT contain a
- sqrt(expectedSrdDev) > 0

Test Input 8:

- PertNetwork has ONE MilestoneNode(n)
- n has ONE inArrow(a)
- a.earliestFinish > 0
- n has ONE inArrow(a)
- n has ONE precedent(p)
- p contains a
- sqrt(expectedSrdDev) < 0

INFEASIBLE

2.3 PertNetwork – getArrowNetwork()

```
1   // Class:   PertNetwork
2   public static PertNetwork getArrowNetwork(ArrayList<Activity> activities){
3
4       PertNetwork an = new PertNetwork(activities);
5
6       // This is the default
7       an.start.setExpectedDate(0);
8
9       // All unconditioned activities leave the start node
10      for(Activity a : activities){
11          if(a.getPrecedents().size() == 0){
12              an.start.addOutArrow(a);
13          }
14      }
15
16      // All activities without dependents feed into the final node
17      for(Activity a : activities){
18          if(a.getDependents().size() == 0){
19              an.finish.addInArrow(a);
20          }
21      }
22
23      // Recursively connects the graph
24      an.linkNodeForward(an.start);
25
26      // Now we know the last node number and set it
27      an.finish.setName(index);
28
29      an.forwardPass();
30
31      return an;
32  }
```

Figure 5: getArrowNetwork() - Code

```
Class PertNetwork
private void getArrowNetwork()
```

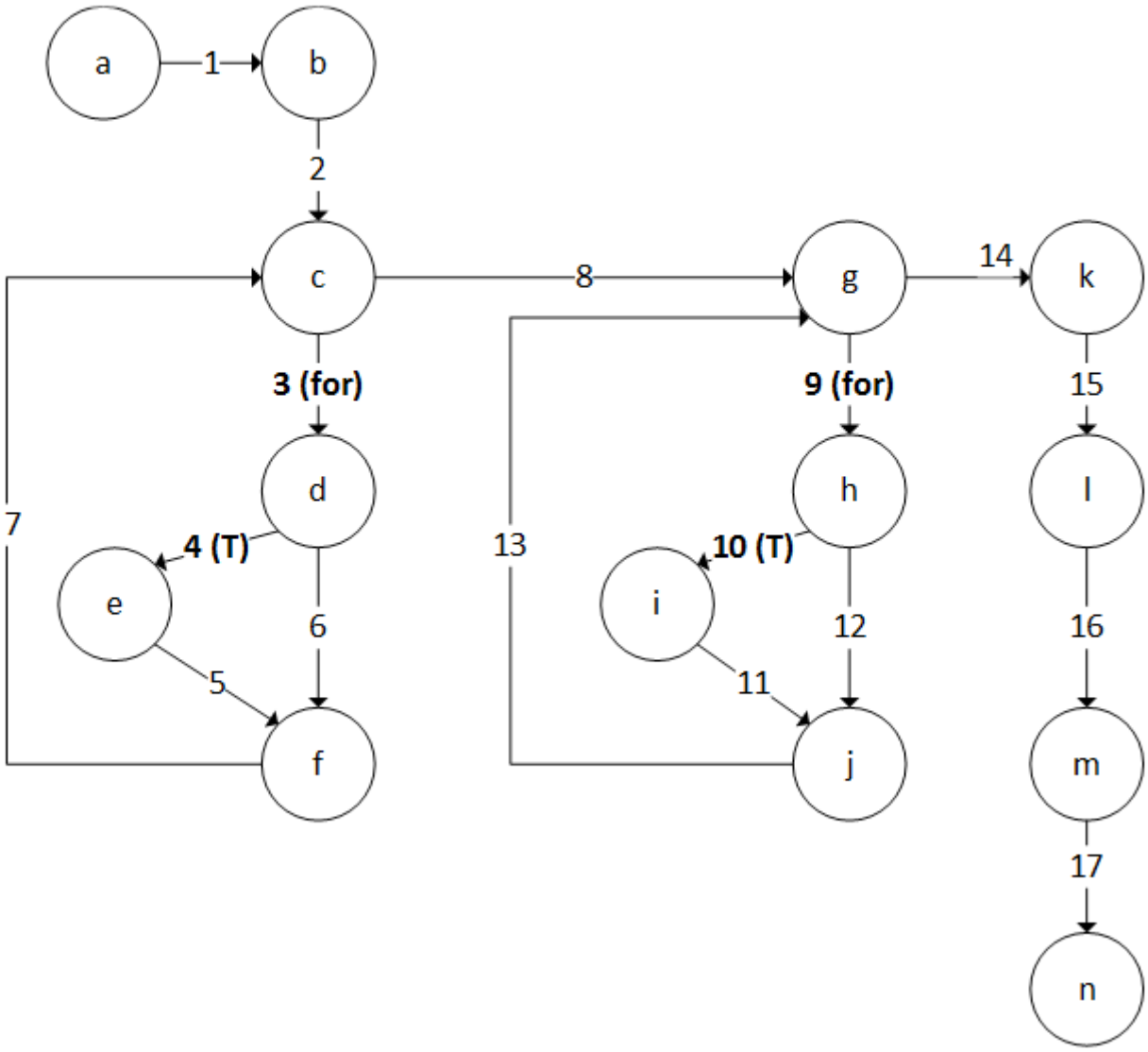

Figure 6: backwardPass() - CFG

Cyclomatic-Complexity:

Complexity = num. Decision nodes + 1

= 4 + 1

= **5**

Basis-Path Coverage:

Test Cases: {

1: <1,2,3,4,5,7,8,9,10,11,13,14,15,16,17>

2: <1,2,8,9,10,11,13,14,15,16,17>

3: <1,2,3,6,7,8,9,10,11,13,14,15,16,17>

4: <1,2,3,4,5,7,8,14,15,16,17>

5: <1,2,3,4,5,7,8,9,12,13,14,15,16,17>

}

Test 1 Input:

- List of ONE Activity(a)
- a has no precedents
- List of ONE Activity(a)
- a has no precedents

Test 2 Input:

- List of ONE Activity(a)
- List of NO activities

INFEASIBLE

Test 3 Input:

- List of ONE Activity(a)
- a has precedents
- List of ONE Activity(a)
- a has no precedents

INFEASIBLE

Test 4 Input:

- List of ONE Activity(a)
- a has no precedents
- List of NO activities

INFEASIBLE

Test 5 Input:

- List of ONE Activity(a)
- a has no precedents
- List of ONE Activity(a)
- a has precedents

INFEASIBLE

2.4 EarnedValue – calculateActualCost()

```
1   // Class:   EarnedValue
2   private void calculateActualCost() {
3
4       double ac = 0;
5
6       for (Activity a : project.getActivityList()) {
7           if (a.getStatus()) {
8               ac += a.getActualCost();
9           } else {
10              if (a.getPercentComplete() > 0) {
11                  ac += a.getActualCost() * a.getPercentComplete();
12              }
13          }
14      }
15
16      project.setActualCost(ac);
17  }
```

Figure 7: calculateActualCost() - Code

```
Class EarnedValue
private void calculateActualCost()
```

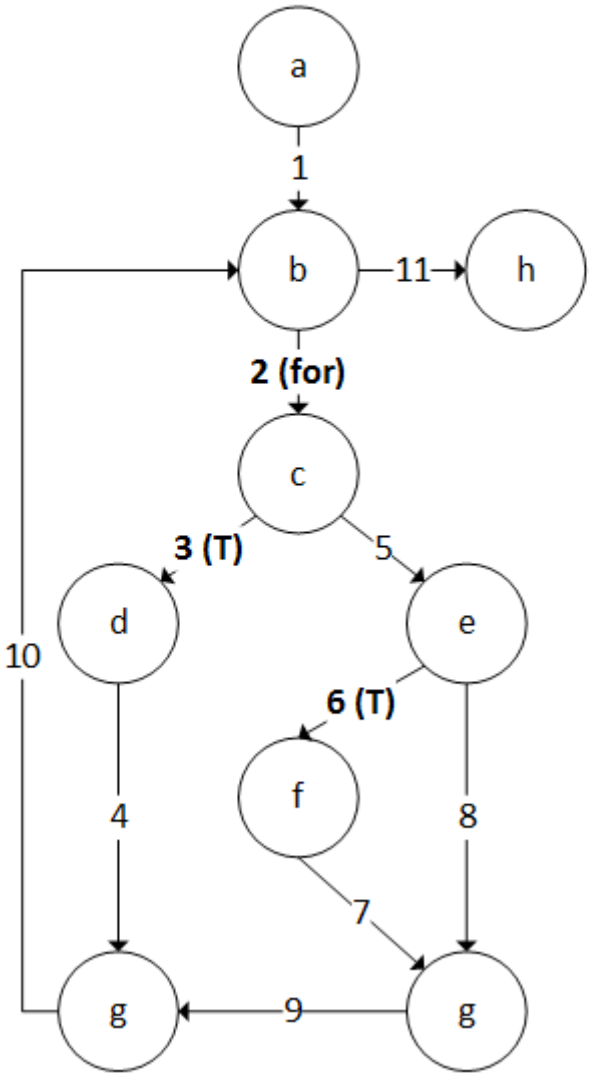

Figure 8: calculateActualCost() - CFG

Cyclomatic-Complexity:

Complexity = num. Decision nodes + 1

= 3 + 1

= **4**

Basis-Path Coverage:

Test Cases: {

1: <1,2,3,4,10,11>

2: <1,11>

3: <1,2,5,6,7,9,10,11>

4: <1,2,5,8,9,10,11>

}

Test 1 Input:

- Project(p) has ONE Activity(a)
- a.status = true

Test 2 Input:

- Project(p) has NO Activities

Test 3 Input:

- Project(p) has ONE Activity(a)
- a.status = false
- a.percentComplete > 0

Test 4 Input:

- Project(p) has ONE Activity(a)
- Project(p) has ONE Activity(a)
- a.status = false
- a.percentComplete = 0

2.5 MilestoneNode – toStringArrows()

```
1   // Class:   MilestoneNode
2   public String toStringArrows(String type) {
3
4           String  arrowString = "";
5           ArrayList<Activity> arrows;
6
7           if(type.equals("in")){
8               arrowString = "Activities to Complete by this Milestone: ";
9               if(this.hasInArrows()){
10                  arrows = this.getInArrows();
11              }
12              else{
13                  arrows = new ArrayList<Activity>();
14                  return "No in arrows";
15              }
16
17          } else if(type.equals("out")){
18              arrowString  = "Activities to Available to start after this Milestone: ";
19              if(this.hasOutArrows()){
20                  arrows = this.getOutArrows();
21              }
22              else{
23                  arrows = new ArrayList<Activity>();
24                  return "No out arrows";
25              }
26
27          } else {
28              ec.showError("You didn't enter 'in' or 'out' when you should've.");
29              return null;
30          }
31
32          for(Activity a : arrows){
33              arrowString += a.getName() + ", ";
34          }
35          if(arrowString.endsWith(", ")){
36              arrowString = arrowString.substring(0, arrowString.length()-2) + "\n\n";
37          }
```

Figure 9: toStringArrows() - Code

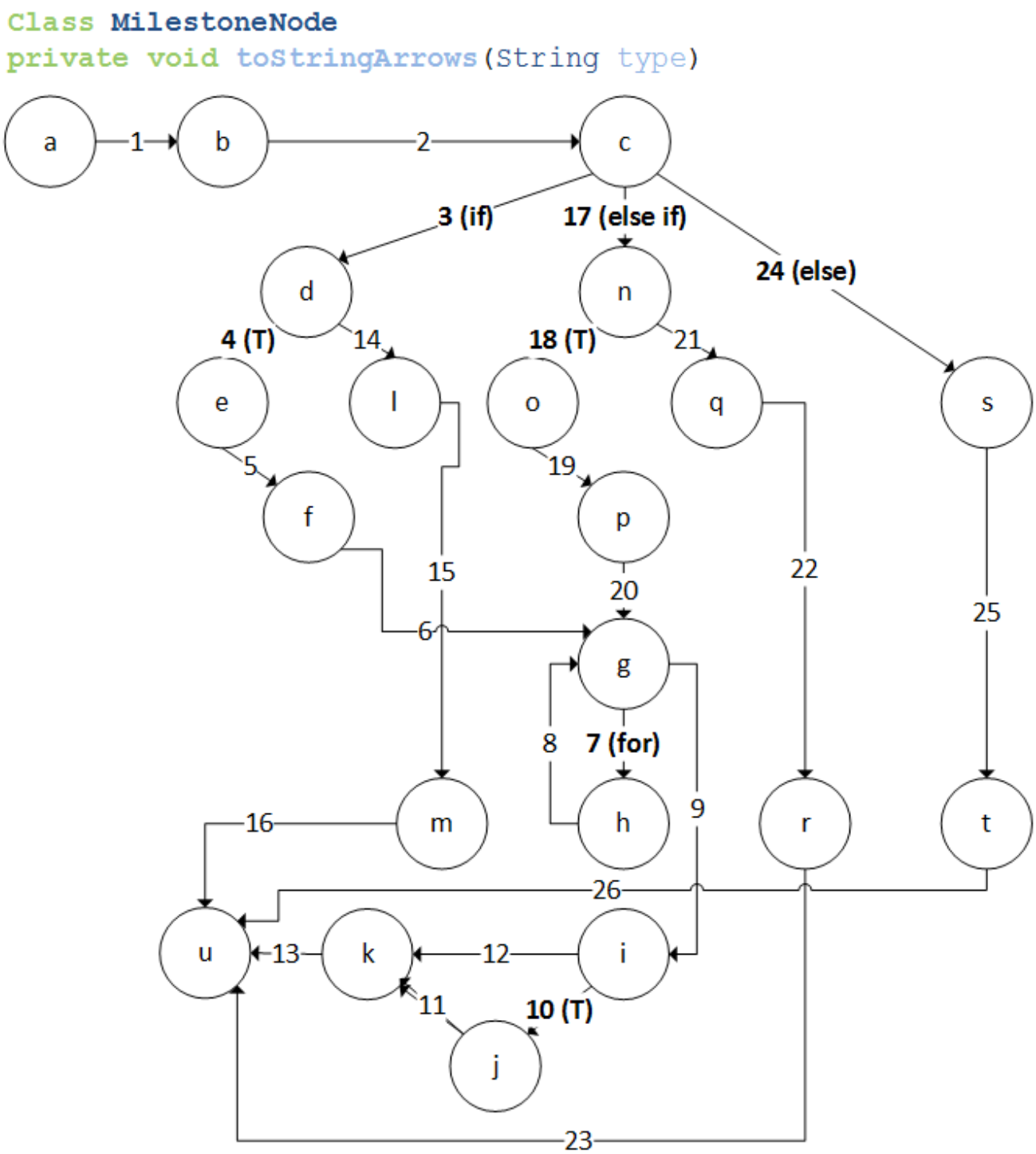

Figure 10: backwardPass() - CFG

Cyclomatic-Complexity:

Complexity = num. Edges – num. nodes + 2

= 26 – 21 + 2

= **7**

Basis-Path Coverage:

Test Cases: {

1: <1,2,3,4,5,6,7,8,9,10,11,13>

2: <1,2,3,4,14,16>

3: <1,2,3,4,5,6,9,10,11,13>

4: <1,2,3,4,5,6,7,8,9,12,13>

5: <1,2,17,18,19,20,7,8,9,10,11,13>

6: <1,2,17,21,22,23>

7: <1,2,24,25,26>

}

Test 1 Input:

- type = "in"
- MilestoneNode has ONE inArrow
- MilestoneNode has ONE inArrow
- arrowString ends with ", "

Test 2 Input:

- type = "in"
- MilestoneNode has NO inArrow

Test 3 Input:

- type = "in"
- MilestoneNode has ONE inArrow
- MilestoneNode has NO inArrows

INFEASIBLE

Test 4 Input:

- type = "in"
- MilestoneNode has ONE inArrow
- MilestoneNode has ONE inArrow
- arrowString DOES NOT end with ", "

INFEASIBLE

Test 5 Input:

- type = "out"
- MilestoneNode has ONE outArrow
- MilestoneNode has ONE outArrow
- arrowString ends with ", "

Test 6 Input:

- type = "out"
- MilestoneNode has NO outArrows

Test 7 Input:

- type not equal "in" or "out"

3.0 Black-Box Testing

Worst-case boundary-analysis testing requires that for each input variable we determine the 5 following values:

- **min** – the minimum value
- **min+** – a value just above minimum
- **nominal** – that average value
- **max-** – slight below the maximum
- **max** – the maximum value

As worst-case analysis rejects the single-fault assumption, we must then test the function with every possible combination of the above . This results in **5n** tests to run (where n is the number of parameters to the function).

For each function, a table containing these 5 values for each parameter is provided. These inputs are based on the ranges specified in section 1.2. The tests must be run for each combination.

3.1 EarnedValue - getActivityScheduleValue()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
ActivityEarlyFinish	0.0	1.0	30.0	730.0	731.0
activityLateFinish	0.0	1.0	30.0	730.0	731.0
activityPlannedVal	0.0	1.0	30.0	730.0	731.0
activityDuration	0.0	1.0	30.0	730.0	731.0
daysSinceStart	0.0	1.0	30.0	730.0	731.0

3.2 Project - areValidPercentages()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
Percentage1	0.0	0.01	0.5	0.99	1.0
Percentage2	0.0	0.01	0.5	0.99	1.0

3.3 Project - areValideValues()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
BudgetAtCompletion	0.0	1.0	1000.0	9999999.0	10000000.0
ActualCost	0.0	1.0	1000.0	9999999.0	10000000.0
EarnedValue	0.0	1.0	1000.0	9999999.0	10000000.0

3.4 Activity - areValideValues()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
MostLikely	0.0	1.0	30.0	730.0	731.0
Optimistic	0.0	1.0	30.0	730.0	731.0
Pessimistic	0.0	1.0	30.0	730.0	731.0

3.5 Activity - areValidPercentAndCost()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
PercentComplete	0.0	0.01	0.5	0.99	1.0
ActualCost	0.0	1.0	500000.0	999999.0	1000000.0

4.0 Test Headers

Below, the **Test Headers** for each implemented test is provided. The information they include are:

- Identifier:
- Brief description:
- Preconditions:
- Inputs and expected outputs:
- Expected post-conditions:

4.1 White-Box Tests

For these tests, one function for each of the feasible paths was written. How the input was constrained is specified in each test header.

4.2 Black-Box Tests

Only one black-box test was written for each function in this case, as unlike our white-box implementations, it was much simpler to iterate through all test cases automatically.

5.0 Discussion

Blabs blabsblabs

References:

Daniel Sinnig PhD: *Lecture Slides, COMP 354*