CURTIN UNIVERSITY (CRICOS number: 00301J)

Department of Computing, Faculty of Engineering and Science

Data Structures and Algorithms (COMP1002)

PRACTICAL 0 - PLANNING

OVERVIEW:

Throughout this semester, we will be using the practical sessions to develop a full application that has to use a variety of data structures. The application is an implementation of part of the research produced by a PhD thesis relating to network reliability. It shows the way a number of data structures and algorithms can work together in a moderately large program. This practical is all about taking the system requirements described in this document and creating a design that you will then implement and test over the coming weeks.

Note that the work you undertake in the worksheets after this one will need to be submitted electronically to Blackboard, but this practical **does not** have to be submitted. The work in many of the worksheets including this one will form part of your assignment, so falling behind in the worksheets or not completing them properly will have implications on the marks you get for that assessment task. In particular, not making a sufficient attempt at the assignment results in automatically failing this unit.

AIMS:

- To practice the design of an Object-Oriented program.
- To design the program you will be creating for the assignment in this unit.

BEFORE THE PRACTICAL:

- Read this practical sheet fully before starting. You may wish to review your notes from Object Oriented Program Design.
- Look ahead to the Practical 01 worksheet, since that is where you will start to implement the planning you do in this session.

ACTIVITY 1: SYSTEM REQUIREMENTS SPECIFICATION

The system that we will develop this semester implements an algorithm that computes reliability measures of a network. In particular, it computes a reliability figure (between 0

and 1) that measures the chance that a network can be used as intended at any given time. If a network has a low reliability, it may be necessary to change its design by adding additional routes to ensure that communications can be established.

The first part of the specification is the network to be analysed. This will generally be read in from a file, and can be expressed as a mathematical graph. Such a graph consists of *vertices* and the *edges* that link the vertices. Each edge and vertex represents a physical component such as a wired connection or a router. Each component has its individual reliability (a figure between 0 and 1) that measures the chance of it being available in any given time period. Most components will have a reliability very close to 1, but for simulations it is usual to use a reliability of 0.9 for each component to allow for comparisons with other work.

Our implementation will read the reliability for each component in from a file along with the rest of the information on that component; this information includes the name of the component (an optional string), the type (another string which is often “router” or a cable specification such as “CAT5”) and the cost (in US$). For components that will be represented by edges the adjacent components (both represented by vertices) are also read in from the file.

The network needs to be stored in an appropriate structure that makes access during the running of the algorithm as easy as possible. The network will first be sorted (to increase the performance of the algorithm) and then processed one component at a time in order to compute the overall reliability of the network. For this unit we will assume that edges are undirected (*i.e.,* communication can pass along an edge in either direction). The main purpose of the algorithm is to compute the reliability for the network as a whole. We can assume that the reliability is the probability that one chosen vertex (called the *source*) is connected to a second chosen vertex (called the *target* or *sink*).

The processing takes the form of a decision diagram; each level of the diagram represents the decision on the state (available or failed) of a component. The diagram starts at one *node* (representing communication having reached only the source vertex) and processes it to produce two child nodes; one of these represents the state for which the component in question has failed and the other represents the component being available. Each node must store the probability of being in the state of the network that the node represents as well as information on that state.

The way that the network state is encoded is a node is through the use of *blocks*. These store vertices that are known to be connected. From this, we can compute whether the vertices that we care about are connected, and can thus communicate with each other. More details on blocks will be given in later practicals.

During processing, the nodes on the level currently being processed are stored in Q_C, the queue representing unprocessed nodes and the current level of the diagram. Child nodes are added to the next level, Q_N. One of the important aspects of this method is the concept of *isomorphism*, when a new child node is created it has to be compared with all child nodes already on Q_N. If an existing child is found to have an identical network state, the new node is merged with the existing one; if not the new node is added to the end of Q_N. This is repeated until Q_C is empty, at which point the contents of Q_N are moved to Q_C for processing.

Some nodes have special status as *terminal nodes*. These are nodes for which we can determine that they represent a state where the communication through the entire network has either reached its goal (a *success node*) or is unable to reach its goal (a *failure node*). When a child node is found to be terminal it is not added to Q_N and hence is not processed further.

The algorithm finishes when both Q_N and Q_C are empty. At this point the reliability of the network should be displayed, normally to 8 decimal places but possibly up to 20.

Note: For this unit you will not have to write the code to perform node state manipulations and some other parts of this work; these classes and/or methods will be given to you when needed.

ACTIVITY 2: DESIGNING THE SYSTEM

Take the following steps. At each step, it may be useful to first identify the choices that need to be made. When you have considered what the choices are, discuss them with one or more of the people near you. Ideally you want to make each choice because there is an obvious advantage to making that choice, or a disadvantage to choosing the other options.

1. Identify the classes. These are generally nouns such as Edge or Node. If you end up with nouns that don't have any actual relevance to the system, remove them from your list.
2. Identify any inheritance relationships or even if inheritance makes sense. Also consider other relationships between classes.
3. Fill in class details by adding class fields.
4. Decide on what actions each class needs to be able to perform; these actions will be the methods of the class. Your classes should generally have creators, accessors and mutators.

You can find a UML diagram for a slightly more complicated version of this problem on Blackboard. This diagram was created by a team for final year students who implemented that version of the problem for their Software Engineering Project work. They won the prize for best project team (as can be seen on the mirrored plaque on the 3rd floor of building 314) which should give some indication that they did good work. Note that this is a first-

level UML diagram since it includes the connectivity between classes but not the details of each class.

This diagram will be different from your own because it was designed to be more general – in particular it has interfaces a number of interfaces to allow certain aspects of the implementation to be extended more easily at a later date. You can also ignore anything not coloured green, since that refers to the system they used (including a graphical tool). You may also note that they refer to Q_N and Q_C as DAQueue and SQQueue respectively, for reasons of the way that those structures are used which we haven't covered yet.

Compare your class diagram with one or more people near you. If there are differences, try to discover the reasons behind them. In some cases these differences are choices that can legitimately be made either way, although some may be errors. If in doubt, ask your tutor.

ADDITIONAL INFORMATION

The information above covers the part of the project that you need to consider for next week's lab. This section gives more information on the classes needed for the problem. This isn't formally part of the practical, but has been added since the information will become relevant later in the semester.

One of the two top-level classes has already been mentioned above; the Network. The other such class is the Diagram, which contains all of the nodes. The Diagram also contains Q_C and Q_N and a running total of the reliability (and optionally, unreliability – more on this later) of the diagram so far (a float). It depends on the Network for information on Edges and Vertices used in the processing of parent nodes into child nodes. In the version of the decision diagram being used (the Hybrid Ordered Binary Decision Diagram), the nodes are never explicitly linked into a tree structure, although other versions do this. Once a node has been processed it is discarded.

Each child node stores a reliability score for the state that it represents as well as storing the network state itself. The first node of the diagram (the root node) is created with a default state and a reliability of 1.0 since no components have had a chance to fail yet. When processed, a node is processed into a positive child node (representing that the component in question is available) using the node's *contract* method (which will be provided to you) and a negative child node (representing a failed component) using the *delete* method.

Both of these methods effectively create a new child node, which is then stored in Q_N. Theoretically the parent node is deleted once the processing is finished, but it is more efficient to use the delete method to transform the parent into the child rather than creating a new node and deleting the old one. This means that while Node.contract returns a new Node, Node.delete actually changes the existing node and doesn't return anything. You don't need to worry about this process, but feel free to ask about if or look at the provided code (when available) if you're curious.

The child node has a reliability that is based on that of the parent, but modified by the decision on the component. Thus Node.contract multiplies the reliability of the parent by the probability that the component is available, while Node.delete multiplies by the probability that the component is not available. Since we assume that these are the only two possibilities, Pr(not available) = 1 – Pr(available), and hence components only need to store one reliability.

When a child node is found to have a state that represents a successful communication (*i.e.,* a communication path exists from the source to the target) it is called a success node, and isn't added to Q_N. Instead its reliability is added to the reliability recorded in the Diagram, which initially starts at 0. When the algorithm completes, this reliability is the reliability of the network.

When a child node is found to have a state that means that a successful communication isn't possible it is called a failure node. It is treated like a success node, except that its reliability isn't added to the Diagram reliability; if an unreliability score is being kept it is added to that instead, otherwise it is discarded. An unreliability is no use other than as a check – at the end of the process the reliability and the unreliability should add up to 1, which is a useful check to see if the process has completed properly.

The network state is encoded in a node as blocks. Each block is a collection of vertices that are connected in the current state. Examples of this will be covered in the lecture of week 2, but a network state could look like [1 3][2]*[4] or [1 2 4]*[3]. The details of this don't matter terribly for your work, but in general a block such as [1 3] means that vertices 1 and 3 are connected. Similarly [1 2 4] means that these three vertices are all connected. Because we are assuming that communication is undirected, this means that you can reach any of these vertices from any of the other ones. The asterisk (*) indicates which block is connected to the source vertex; this block is called the *marked* block. If the target vertex is ever found in marked block then we know that the source and target are connected, and the node is successful.

The trick is that the numbers in these blocks don't actually refer to the actual vertices, but only to something called the *active vertices*, and explaining those gets a bit complicated (but I'll try in the lecture). When a vertex stops being active it is deleted from all of the blocks – essentially we no longer care about it because it no longer provides useful information. This means we're more likely to have more *isomorphism* happening.

When a node has been created, its state is compared to the state of all nodes on Q_N. If the two states are equal (in other words the blocks are the same and the same block is marked) then the two nodes are isomorphic and are merged. In practical terms, this means we add the reliability of the new node to the reliability of the node already on Q_N. We don't need the rest of the information since it's the same for both nodes anyway.

This isomorphism reduces the exponential growth of nodes, and is the key part of the use of Ordered Binary Decision Diagrams. Normally a diagram that would test 100 components would end up with 2^{100} nodes on the bottom row, but using isomorphism we can reduce this to a very small amount. In fact, the bottom row of our diagram will always have two nodes – one success and one failure – and we don't store either one of these. Practically speaking this means that we can compute the reliability of a fairly large network (say a 5×15,000 vertex network laid out in a grid) and rather than getting $2^{75,000}$ nodes at the end, we get a maximum of 90 rows on any one level. Because we don't store nodes once they have been processed, this means never storing more than around 180 nodes instead of storing all 12,147,646 generated. As you can see, this is a useful improvement in efficiency.

SUBMISSION DELIVERABLE:

This worksheet **is not** to be submitted separately to Blackboard. It will form part of your assignment.