Automatic Graph Generation in a Unix Environment.

Lory D. Molesky
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

COINS Technical Report 89–122
July 5, 1989

This research supported in part by the National Science Foundation under grants MCS-8306327, DCR-8318776, and DCR-8500332.

Automatic Graph Generation in a Unix Environment.

Lory D. Molesky

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

Two utilities for generating and visualizing graphs are presented. Randog generates random graphs for use in multiprocessor scheduling simulations. Xgraph generates a plot of a graph from an adjancency list. Both tools are written in C, and run on various types of Unix machines.

This research supported in part by the National Science Foundation under grants MCS-8306327, DCR-8318776, and DCR-8500332.

Contents

List of Figures

This technical report describes two utilities for graph generation. Xgraph constructs a layout of a graph from an adjacency list. Randog generates random directed acyclic graphs. Both programs are written in the C programming language. The Randog program runs on DEC Microvaxes and Sequent Symmetries, Xgraph runs on any machine supporting X windows (version 10).

Section 1 describes Randog, section 2 describes Xgraph. Throughout both sections of this document, graphs generated with Xgraph are used for illustrations.

1. Randog

1.1 Introduction

Randog (RANDOm Graph) was developed for use in multiprocessor scheduling experiments. Researchers wishing to use simulation techniques to evaluate the performance of real-time scheduling algorithms can use Randog for generating their input task sets. Randog generates a periodic *task graph* – an arrangement of tasks with precedence constraints and deadlines. Vertices of the output graph represent tasks, while an edge between two tasks represents a communication path.

The directed edges impose a partial order on the task graph. The existence of a communication path from a task *a* to a task *b* implies that task *a* must complete its execution prior to the execution of task *b*. The terms communication path, edge, and precedence constraint are used interchangeably throughout this document.

Task computation times, task communication times and deadlines impose timing constraints on the graph. Each task in the graph has a computation time, while each directed edge has an associated communication cost. Some tasks in the task graph will have deadlines. Experiments performed in [4] illustrate the use of such precedence constrained graphs in the evaluation of scheduling algorithms. The terms vertex, node, and task are used interchangeable in this document.

The Randog user supplies a data file describing certain attributes of the task graph. Some graph properties, such as the total number of tasks, are precisely specified by the user. Other properties are defined as a probability. For instance, the computation time of all tasks is specified as a range, then for each task, a random value is drawn from this range over a uniform distribution.

Elementary groups are the building blocks of a task graph. The Randog user specifies characteristics of the elementary group, such as the number of tasks and the general interconnection scheme. Elementary groups are replicated to facilitate the imposition of periods at regular intervals in the *task graph.*

Section 1.2 describes the topology of *task graphs.* Sections 1.3 and 1.4 describe the construction of *elementary groups.* Section 1.5 describes the periodic relationships in *task groups*, illustrating example Randog output. Section 1.6 summarises Randog.

1.2 Topology of Task Graphs

In order to understand the potential range of Randog graph outputs, it is necessary to understand the topology of the task graph. *Elementary groups* are the basic building blocks of task graphs. The shape and size of elementary groups are specified by the user. To impose periodic relationships on the task graph, Randog constructs *Periodic Task Groups* (PTG's) from elementary groups. A PTG contains one or more elementary groups and has a deadline at the leaf nodes (the leaf nodes of the last elementary group in the PTG). *Task groups* are constructed from one or more PTG's. Finally, Randog constructs the *task graph* by composing task groups.

Fictitious start and end vertices are added to the top and bottom of the task graph. These vertices serve only to connect the various task groups into a single graph. A fictitious vertex contains no resource constraints; this vertex (task) has a computation time of 0, and all edges to and from this vertex have a communication cost of 0. Figure 1 illustrates the relationship between the elementary group, task group, and the task graph. The vertices inside dashed rectangles are the elementary groups of the graph. Vertices inside solid rectangles are the task groups.

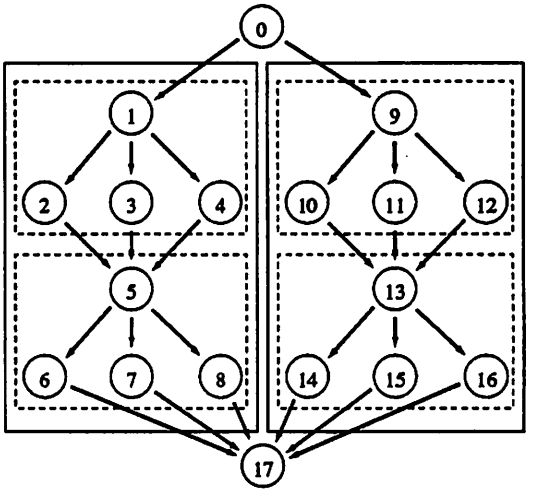

Figure 1: Structure of a Task Graph

Randog constructs a task graph which contains N different task groups. N, the number of task groups in the task graph, is specified by the user. PTG's are composed of elementary groups. Task group i has $N - i + 1$ PTG's, where each PTG is composed of $lcm(1, 2, ..., N)/(N - i + 1)$ elementary groups. Computing the $lcm(1, 2, ..., N)$ allows the construction of N task groups with periods (1,2,3,...N). For instance, if the user specifies 2 task groups, the $lcm(1, 2)$ is computed, yielding 2. Task group 1 has (2 - 1 + 1), or 2 PTG's. Task group 2 has (2 - 2 + 1), or 1 PTG's. PTG1 is composed of 1 elementary group, while PTG2 is composed of 2 elementary groups.

The example illustrated in figure 1 shows how a task group is generated from elementary groups.

parameter	type	description
number-task-groups	integer	The number of task groups in the task graph.
task-group-type	integer	This integer represents *chain*, *general*, *rtree* or *tree*.
group-size	integer	Number of tasks per group.
layer-size-ratio	float	The fractional metric of tasks per layer.
successors-per-task	float	Mean successors for each task.
cross-edge-fraction	float	Probability that a task has an edge to a distant layer.
comp-low comp-high	integer	Defines the range of computation values per task.
comp-fraction	float	Scales the computation value assigned to an edge.
redundancy-no	int	The maximum duplication of a task.
redundancy-pct	float	Percentage of tasks which are redundant.
laxity-factor	float	The slack in defining the base period.

Table 1: Randog input parameters

The elementary groups contained in the task graph are all of type *rtree* (reverse tree), instances of elementary groups are vertices {1 2 3 4}, {5 6 7 8}, {9 10 11 12}, and {13 14 15 16}. Task group 1 is composed of 2 PTG's (verticies {1 2 3 4} and {5 6 7 8}, each PTG is composed of 1 elementary group. Task group 2 is composed of 1 PTG (verticies {9 10 11 12 13 14 15 16}). This PTG is composed of two elementary groups (verticies {9 10 11 12} and {13 14 15 16}). Task group 1 will have 2 periods, the first deadline will be at tasks {2 3 4}, the second deadline at tasks {6 7 8}. Task group 2 has 1 period, the deadline at tasks {14 15 16}.

Summarizing the topological hierarchy, a *task graph* is composed of *task groups*. *task groups* are composed of *elementary groups*. An additional, semantic relationship, is imposed between *task groups* and *elementary groups*, the PTG.

1.3 Elementary Groups

While we have discussed how Elementary groups are replicated to form a task graph, we have yet to discuss what constitutes an elementary groups. Randog requires that the user specify the type (*task-group-type*) of the elementary graph to be used in each task group. The user also specifies the number of tasks in an elementary group, the *group-size*. Table 1 illustrates all Randog input parameters.

The user has a choice of the types of elementary groups – *chain*, *general*, *rtree*, or *tree*. The elementary group type defines the interconnection structure of communication paths between tasks (edges between vertices). If task a has a communication path to task b, then task b has a as its predecessor. Similarly, task a has successor b. For all tasks contained in an elementary group, the properties listed in table 2 hold.

The type determines the topology of the elementary group. In figure 1, the dotted rectangles

elementary group type	*definition*
rtree	At most one predecessor exists.
tree	At most one successor exists.
general	At least one successor exists.
chain	Exactly one successor exists.

Table 2: Types of Elementary Groups

represent instances of elementary groups. A task group is composed of the same type of elementary group. Moreover, the topology of the elementary group is replicated per task group. Examining figure 1, you will notice that the both task groups happen to be composed of the elementary group of type rtree.

All elementary groups in a task graph will contain the same number of tasks. This restriction is imposed in order to compute a *base period* which is uniform between every elementary group, thus facilitating the construction of periods. Although there are many variations of an elementary group type, Randog generates only one variation per graph. This can be seen in figure 1.

While the communication paths connecting the tasks in an elementary group are replicated in a task group, other attributes are independent among replications. Task communication times and computation times are drawn from a uniform distribution over a specified range. Since the ficticious start and end nodes serve only to connect task groups, they do not have resource constraints. The fictitious start and end nodes are assigned zero computation time. Edges connected to both the fictitious start and end nodes have zero computation time. Edges emanating from leaf tasks of a PTG have communication time of zero. Figure 2 illustrate instances the four different types of elementary groups.

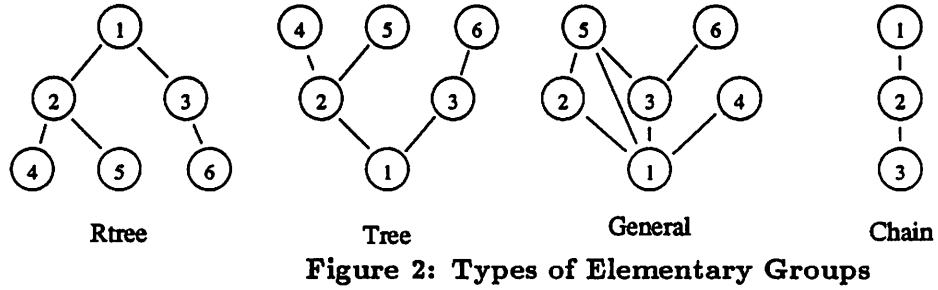

Figure 2: Types of Elementary Groups

1.4 Construction of Elementary Groups

The construction of the elementary groups is performed by using constant and probability parameters. Constant parameters relating elementary groups are *group-size*, the number of vertices in each elementary group, *task-group-type*, the type of the elementary group. The probability parameters which effect the interconnection of the tasks in the elementary group are `successors-per-task`, `layer-size-ratio`, and `cross-edge-fraction`. The probability parameters along with the *task-group-type* guide the placement of the edges in each elementary group.

The task-group-type (*tree, rtree, general* or *chain*) is specified by the user, but the exact placement of the edges is based on the probability parameters. The scheme for interconnecting tasks is to first partition the tasks into layers, then attach edges between layers. In addition to the task-group-type, the attributes *successors-per-task* and *layer-size-ratio* also contribute to the topology of the elementary group. More formally, there are *group-size* vertices in each elementary group, each vertex representing a task. The level, l_i, of task i is defined to be the shortest path length from the root vertex to task i. A layer consists of all the vertices of a particular level.

The partitioning of groups into layers is based on the proper fraction `layer-size-ratio`. To determine how many vertices will comprise each layer, a random integer is drawn with mean `group-size` / `layer-size-ratio`. This procedure is repeated until all vertices in the group have been assigned to a layer. Prior to running the randomized procedure for defining the layer size, one vertex is reserved as a root vertex. For graphs of type tree or general, this vertex is the top vertex.

If a graph is of type tree or rtree, the layers will be sorted. Graphs of type general do not have their layers sorted. A tree will have the layer with the most tasks as the bottom layer, and the least tasks as the top layer. An rtree appears as an upside-down tree. Chains have only one task per layer. Figure 1.3 illustrates these different elementary group topologies.

Between levels, edges are randomly generated based on the mean of an input parameter, `successors-per-task`. For each layer of the graph, each task is connected to the next layer on the basis of this parameter. Note that trees and rtrees are only connected between layers [1].

Although tasks of elementary group type tree and rtree are wired only between layers, general graphs can also be interconnected across layers. The input parameter *cross-edge-fraction*, specifying the probability that a particular task contains an edge to another task in a "distant" layer. This construction creates a more general graph.

Figure 3 illustrates an rtree of `group-size` 6, `layer-size-ratio` .4, and `successors-per-task` of 2. Vertex number 1 resides at level 1, vertices 2 and 3 at level 2, and vertices 4, 5, and 6 are

[1] Although the successor-per-task make sense when applied to a tree, the term successors-per-task is a bit of a misnomer – it really means predecessors per task.

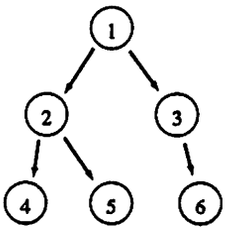

Figure 3: Generating an Elementary Group

in level 3. When partitioning the tasks into layers, the value (`group-size * layer-size-ratio`) is rounded to the nearest integer. This integer is used as the mean of the uniform distribution. Random integers are generated with this mean, allocating tasks to a layer from the task pool until no tasks remain. In the example of figure 3, the mean successors per task is $6 * 0.4 = 2.4 = 2$.

The parameter `group-size` specifies the number of vertices in every group. This convention enables the construction a base period which is applicable to any elementary group. For instance, if one were to specify 4 task groups, each of a different `task-group-type`, the elementary groups comprising the task groups (`rtree, tree, general` and `chain`) would all contain the same number of vertices.

1.5 Construction of Periodic Relationships in Task Groups

Section 1.1 outlined the basic structure of the task graph. In this section we present the computation of the base-period and its relationship to the overall task graph. We wish to partition each task group of the task graph into i groups, s.t. $1 <= i <=$`number-task-groups`. To achieve this, a base period is defined.

```
average-computation-time = (comp-low + comp-high) / 2;
redundancy-factor = (1 + redundancy-no * redundancy-pct);
base-period = average-computation-time * group-size * redundancy-factor *
                                                    laxity-factor;
```

The base period is the expected worst-case computation time of an elementary group. The product `group-size * average-computation-time` reflects this. The *laxity-factor* stretches this value, allowing more leeway in meeting the deadline. The redundancy-factor provides an additional augmentation of the base-period, – the purpose is to allow more time for the scheduler to service redundant vertices.

Multiple copies of the elementary group are combined to construct the periodic task graph. We compute the least common multiple of the natural numbers to `number-task-groups`. Each task-group contains lcm(1,2,...,`number-task-groups`) copies of the elementary group. We impose periods on each task-group to implicitly define deadlines between groups. These periods are defined so that task-group i has `number-task-groups`$-i+1$) periods.

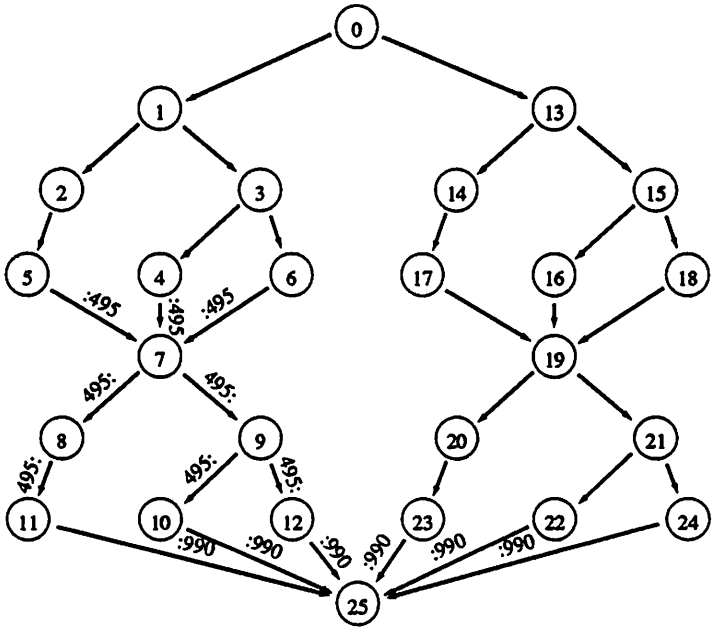

Figure 4: Task Graph with Labeled Deadlines

The example of figure 4 clarifies these relationships. Figure 4 has the following input file:

2 (number-task-groups)
rtree (task-group-type) rtree (task-group-type)
6 (group-size) .35 (layer-size-ratio) 2 (successors-per-task)
0.0 (cross-edge-fraction) 50 (comp-low) 100 (comp-high) .1 (comp-fraction)
0 (redundancy-no) 0.0 (redundancy-pct) 1 (laxity-factor)

In this example there are two task groups of type *rtree*. Notice that each subgraph is the *rtree* of group-size 6 previously illustrated. Lcm(1,2) is 2, so each task group occurs twice in each task group. Task group 2 has one period, task group 1 has two periods This requires that a deadline be imposed on the last task of the first task group of task group 1 (vertex 7), of base-period. A *earliest-start-time* of base-period is also imposed on any successor tasks of vertex 7, (vertices 8 and 9). In the second phase of the period for task group 1 vertices 10, 11 and 12 receive a deadline of base-period * 2. Since task group 2 has a base period length of 1, vertices 22, 23, and 24 receive a deadline of base-period * 2.

Figure 4 illustrates a base period of 495 – task group 2 contains one period, its deadline is 990. task group 1 contains two periods of deadline 495 and 990. The imposition of the base periods on the task graph implicitly define deadlines and earliest-start-times for certain tasks in the task graph. The notation :495 in this figure indicates that the task originating this edge has deadline

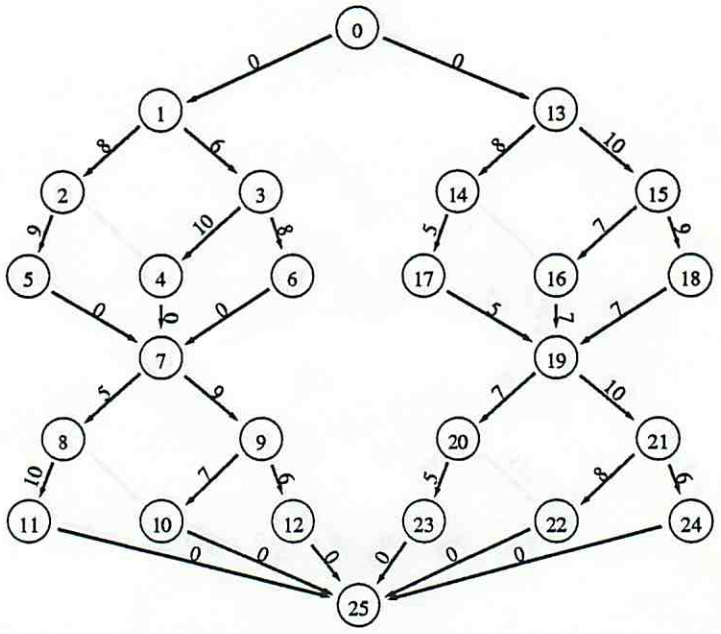

Figure 5: Task Graph with Labeled Communication Times ([5 10])

495. Similarly, the notation 495: indicates that the source task of this edge has a minimum start time of 495. For example in figure 4, the notation :495 on the edge from task 5 to task 7 means that task 5 has a `deadline` of 495. Similarly, the edge from task 7 to task 8 labeled 495: means task 8 has an `earliest-start-time` of 495.

Figure 5 illustrates the same task graph as figure 4, but with different information on the edges. Instead of `deadlines` and `earliest-start-times` placed on the edges, the communication times are illustrated. The `comp-low` and `comp-high` values are 50 and 100, scaled by the `comp-fraction` (.1). (as illustrated in the example input file). The edges of figure 5 are labeled with a random integer selected from this range. Notice that all edges connecting to the dummy vertices (the very top or bottom vertices) have a communication cost of zero. Also notice that, where periods are placed between elementary groups, the communication costs are also zero.

The output of Randog is a precedence constrained task graph with task computation times, inter-task communication times, deadlines, and earliest start times. Table 3 illustrates the output file which contains a description of the generated task graph. One integer, the `deadline` of the entire task graph, is output. This would correspond to 990 in figure 4. For each task, its computation time (`comp`), deadline (`task-deadline`), its earliest start time (`earliest-start-time`), and number of edges (`nconn`) are output. For each edge, the id of the destination task (`connected-to`), and the communication cost (`comm-cost`) are output. All of this information is not contained in any one

parameter	*type*	*applies*	*description*
deadline	integer	per graph	Deadline of the task graph.
number-of-nodes	integer	per graph	Number of vertices in the task graph.
comp	float	per task	Computation time of the task.
task-deadline	float	per task	Deadline of the task.
earliest-start-time	float	per task	Earliest start time of the task.
redundancy		per task	
nconn	integer	per task	Number edges emanating from this task.
connected-to	integer	per edge	Id of the connecting task
comm-cost	float	per edge	Communication cost of this edge.

Table 3: Randog output parameters

graph in this document, but a combination of figures 4 and 5 illustrate most of this data.

1.6 Summary

The Randog program provides an efficient and effective method for generating task graphs which are used as input to real-time scheduling algorithms. This tool is currently being used by University of Massachusetts researchers for the evaluation of real-time scheduling algorithms.

2. Xgraph

The Xgraph program constructs a graph from an adjacency list. Pictorial output is available for several output devices, while printed output is available by printing a generated Postscript file.

The format of the input adjacency list is a series of (*source-vertex destination-vertex label*) triples separated by newlines. The first entry of the adjacency list is the root of the graph. Only vertices reachable from the root will be included in the final output plot.

Xgraph is a general purpose tool for constructing graphs. All the graphs in this document were generated with Xgraph.

2.1 Visual Interfaces

Xgraph can display the final graph on either an X windows display terminal or generate a Postscript file. The dimensions of the graph are controlled by the size of the window containing the graph. This can be specified either on the command line or interactively by manipulating the mouse.

Postscript output will be nearly identical to the X output, except the Postscript output is more visually pleasing. The labels are plotted at the same angle as the edges, the width and font specifiers have more accuracy. The Postscript language for printers [2, 1], is a general purpose graphics programming language, that provides a flexible expressive language for the Xgraph output. Some of the features that Xgraph exploits are scalable fonts, splines, circles, scaling, translation, and rotation. Compared to Postscript, packages for laser printers such as Latex [3] are limited in their capability drawing general purpose graphics.

Xgraph displays vertices as circles. The names of the vertices (the `source-vertex` and `destination-vertex` fields) are centered in the circle. Edges are drawn between vertices, and are optionally labeled (the third entry in the Adjacency list triple). Edges can be directed or undirected.

Command line options exist to control vertex, edge, and font attributes of the graph. One can specify radius of the vertex and the vertex label font. The font for the edge label and the vertex label are also Xgraph options.

When more the one edge exists between a pair of vertices (a multigraph), Xgraph "bends" duplicate edges. The degree of curvature of each additional edge is another Xgraph parameter.

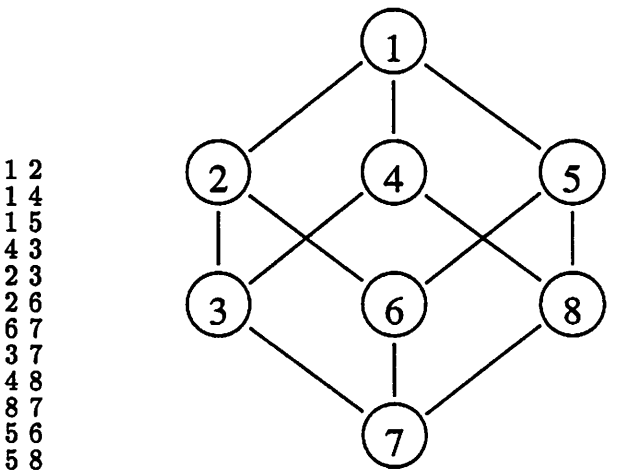

Figure 6: A Cube with Adjacency List Input

2.2 Layout Algorithm

Xgraph performs automatic layouts of a graph. The basic layout algorithm is as follows: vertices are first partitioned into levels. The root vertex is on the first level. The level, l_i, of each remaining vertex i is defined to be the shortest path length from the root vertex to vertex i. Since vertices which are connected to each other are arranged in close proximity to each other, this method of partitioning the vertices in one dimension is a reasonable first pass to minimize edge crossings. The ordering of the vertices per layer will be in the the order specified in the Adjacency list.

Although this strategy is flawless for rooted trees (see figure 8), the general problem of minimizing crossings in NP-hard. An even more difficult problem is that of finding the layout which is most visually pleasing to the user, especially since the concept of what makes a visually pleasing layout may differ between users.

2.3 Summary

The Xgraph tool provides an effective method for generating a graphical layout from an Adjacency list. In addition to working interactively with a graph via an X windows output, Postscript output is also available. The inclusion of these output graphs into technical documents written primarily in Latex is an effective method of constructing expressive documents. All of the figures in this report were generated with Xgraph.

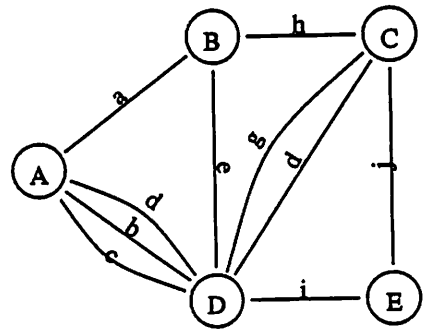

Figure 7: A Directed Multi-graph

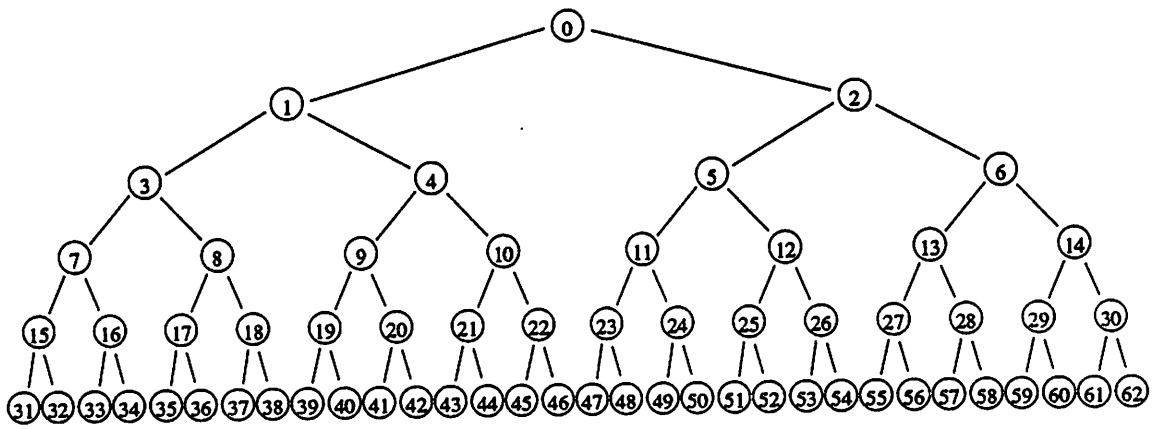

Figure 8: A Complete Binary Tree

References

[1] Adobe Systems Inc. *PostScript Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1985.

[2] Adobe Systems Inc. *PostScript Language Tutorial and Cookbook.* Addison-Wesley, Reading, Massachusetts, 1985.

[3] Digital Equiment Inc. *LaTex users guide & reference manual.* Addison-Wesley, Reading, Massachusetts, 1985.

[4] H. Kasahara and S Narita. Practical multiprocessor scheduling algorithms for efficient paralled processing. *IEEE Trans. on Computers*, C-33(11), 1984.