Appendix B

Graphs and Digraphs

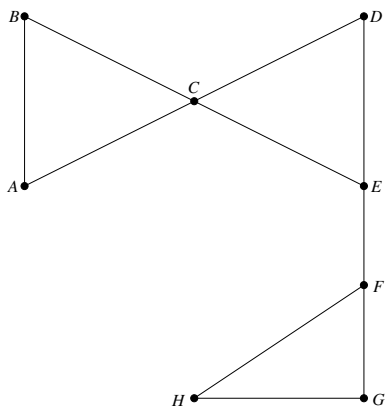
In this thesis we make frequent use of *graphs* to represent various properties of networks of processes. We adopt the terminology of [Wilson 1985].

A *graph* G is defined to be a pair $(V(G), E(G))$, where $V(G)$ is a non-empty finite set of elements called *vertices*, and $A(G)$ is a finite *family* of *unordered* pairs of elements of $V(G)$ called *edges*. (A family is a collection of elements like a set, except that an element may occur more than once; *e.g.* $\{a, b, c\}$ is a set, but (a, a, c, b, a, c) is a family.)

A *digraph* D is defined to be a pair $(V(D), A(D))$, where $V(D)$ is a non-empty finite set of elements called *vertices*, and $A(D)$ is a finite family of *ordered* pairs of elements of $V(D)$ called *arcs*.

A graph (or digraph) is *simple* if there are no duplicate edges (or arcs) uv and no 'loops' uu.

Figure B.1: A Graph



A *walk* in a graph (or digraph) is a finite sequence of edges (or arcs) of the form

$$\langle v_0\, v_1,\, v_1\, v_2,\, ..,\, v_{m-1}\, v_m \rangle$$

A walk in which all the edges (or arcs) are distinct is called a *trail*; if, in addition, the vertices $v_0, v_1, .., v_m$ are distinct (except, possibly, $v_0 = v_m$), then the trail is called a *path*. A path or trail is *closed* if $v_0 = v_m$. A closed path is called a *circuit*.

The simple graph

$$(\{A, B, C, D, E, F, G, H\}, (AB, BC, CD, DE, EC, CA, EF, FH, GH, FG))$$

is illustrated in figure B.1. Here the sequence $\langle AB, BC, CA \rangle$ is both a closed trail and a circuit; the sequence $\langle AB, BC, CD, DE, EC, CA \rangle$ is a closed trail but not a circuit.

A graph with no circuits is known as a *tree*. If D is a digraph, the graph obtained from D by replacing each arc by a corresponding edge is called the *underlying graph* of D. A *directed tree* is a digraph of which the underlying graph is a tree.

A graph is *connected* if there exists a path between any two vertices. The vertices of a disconnected graph may be partitioned into *connected components* such that two vertices are in the same connected component if, and only if, there exists a path between them.

A graph is said to have a *separation vertex* v (sometimes called an *articulation point*) if there exist vertices a and b, where $a \neq v$ and $b \neq v$, and all paths connecting a and b pass through v. In the graph of figure B.1 the separation vertices are C, E and F. A graph which has a separation vertex is called *separable*, and one which has none is called *non-separable*.

Let $V' \subseteq V(G)$. If the induced subgraph $G'(V', E')$ (where E' is the set of edges of G which connect vertices of V') is connected, non-separable and for every larger V'', $V' \subset V'' \subseteq V$, the induced subgraph $G''(V'', E'')$ is separable, we say that V' is a *non-separable component* of G. In the graph of figure B.1 the non-separable components are $\{A, B, C\}$, $\{C, D, E\}$,$\{E, F\}$, and $\{F, G, H\}$.

A *disconnecting edge* of a graph is an edge, the removal of which increases by one the number of connected components. This is also known as a *bridge*. If all the disconnecting edges of a graph are removed the residual connected components are known as *essential components* of the original graph. The graph illustrated in figure B.1 has a single disconnecting edge EF. Its essential components are $\{A, B, C, D, E\}$ and $\{F, G, H\}$.

A digraph is *strongly connected* if, for any two vertices u and v, there exists a path from u to v and also from v to u. The vertices of a digraph which is not strongly connected may be partitioned into *strongly connected components* using the equivalence relation $\sim$, where $u \sim v$ means that there is a path from u to v and also from v to u.

Suppose that the vertex-set of a graph (or digraph) *G* can be partitioned into two subsets V_1 and V_2, such that no edge (arc) joins two elements from the same subset. We say that *G* is *bipartite*.

We denote by $G \setminus e$ the graph (digraph) obtained by removing an edge (arc) *vw*, and combining vertices *v* and *w* into a single vertex (if they are distinct). This is known as an *edge-contraction*. A succession of edge-contractions is called a *contraction*.

The Depth-First Search Algorithm

The Depth-First Search technique is method for scanning the edges (or arcs) of a finite graph (or digraph) which is widely recognised as a powerful technique. It is used by Deadlock Checker in a variety of situations either to perform analysis of transition systems, or to establish global properties of networks, such as the absence of circuits. The algorithm involves constructing a walk which traverses each edge or arc exactly once in either direction.

The algorithms given here are based on those in [Even 1979], where proofs of correctness are to be found.

DFS for Graphs

For a (possibly disconnected) graph the algorithm proceeds as follows. Consider the graph $G = (V(G), E(G))$.

1. Set up two arrays indexed by vertices of $V(G)$: an array of vertices called *father* and an array of integers called *order*. Also set up a boolean array called *used*, indexed by edges of $E(G)$. Set each element of *used* to be *false*, each element of *father* to be "undefined", and each element of *order* to be 0. Also set $i := 0$ and $v := s$ (s is the vertex we choose to start from).

2. Set $i := i + 1$ and *order*$(v) := i$

3. If there are no unused edges incident with v then go to step 5

4. Choose an unused edge $v \xrightarrow{e} u$. Set *used*$(e) :=$ *true*. If *order*$(u) \neq 0$ go to step 3. Otherwise first set *father*$(u) := v$, $v := u$ and then go to step 2.

5. If *father*(v) is defined then set $v :=$ *father*(v) and go to step 3.

6. (*father*(v) is undefined). If there is a vertex u for which *order*$(u) = 0$ then set $v := u$ and go to step 2.

7. (All the vertices have been scanned) Halt.

If we assume a constant time for array lookup then this algorithm can be implemented with linear time. (To implement step 6 efficiently actually requires maintaining a linked list of those vertices that have not yet been visited.)

DFS for Digraphs

For a (possibly disconnected) digraph the DFS algorithm is very similar. Consider the digraph $D = (V(D), A(D))$.

1. Set up two arrays indexed by vertices of $V(D)$: an array of vertices called *father* and an array of integers called *order*. Also set up a boolean array called *used*, indexed by arcs of $A(D)$. Set each element of *used* to be *false*, each element of *father* to be "undefined", and each element of *order* to be 0. Also set $i := 0$ and $v := s$ (s is the vertex we choose to start from).

2. Set $i := i + 1$ and *order*$(v) := i$

3. If there are no unused arcs outgoing from v then go to step 5

4. Choose an unused arc $v \xrightarrow{a} u$. Set *used*(a) := *true*. If *order*$(u) \neq 0$ go to step 3. Otherwise first set *father*$(u) := v$, $v := u$ and then go to step 2.

5. If *father*(v) is defined then set $v :=$ *father*(v) and go to step 3.

6. (*father*(v) is undefined). If there is a vertex u for which *order*$(u) = 0$ then set $v := u$ and go to step 2.

7. (All the vertices have been scanned) Halt.

Checking for Circuit-Freedom of a Digraph

The above algorithm is modified to check for the presence of a circuit in D by maintaining a boolean array, indexed by $V(D)$, to represent which vertices belong to the current *search path*. The digraph has no circuit only if, at step 4, no vertex u is ever found which lies on the current search path.

Finding Non-Separable Components of a Graph

Consider the graph $G = (V(G), E(G))$.

1. Set up three arrays indexed by vertices of $V(G)$: an array of vertices called *father*, and two arrays of integers called *order* and *low*. Also set up a boolean array called *used*, indexed by edges of $E(G)$, and an initially empty stack of vertices, S. Set each element of *used* to be *false*, each element of *father* to be "undefined", and each element of *order* to be 0. Also set $i := 0$ and $v := v_0 := s$ (s is the vertex we choose to start from).

2. Set $i := i + 1$, *order*$(v) := i$, *low*$(v) := i$. Put v on S.

3. If there are no unused edges incident with v then go to step 5

4. Choose an unused edge $v \xrightarrow{e} u$. Set *used*(e) := *true*. If *order*$(u) \neq 0$ then set

$$\textit{low}(v) := \text{Min}(\textit{low}(v), \textit{order}(u))$$

and go to step 3. Otherwise first set *father*$(u) := v$, $v := u$ and then go to step 2.

5. If *father*(v) is undefined or *father*$(v) = v_0$ go to step 9.

6. (*father*$(v) \neq v_0$) If *low*$(v) <$ *order*(*father*(v)) then set

$$\textit{low}(\textit{father}(v)) := \text{Min}(\textit{low}(\textit{father}(v)), \textit{low}(v))$$

and go to step 8.

7. (*low*$(v) \geq$ *order*(*father*(v))) *father*(v) is a separation vertex. All the vertices from S down to and including v are now removed; together with *father*(v) they form a non-separable component.

8. Set $v :=$ *father*(v) and go to step 3.

9. All vertices on S down to and including v are now removed. Together with v_0 they form a non-separable component.

10. If v_0 still has unused incident edges then goto step 12.

11. If there is a vertex u such that *order*$(u) = 0$ then set $v := v_0 := u$ and go to step 2, otherwise halt.

12. Vertex v_0 is a separation vertex. Let $v := v_0$ and go to step 4.

Finding Disconnecting Edges of a Simple Graph

The disconnecting edges of a simple graph are equivalent to its non-separable components of size two. Hence we may find the disconnecting edges of a simple graph, such as a network communication graph, using the algorithm for non-separable components.

Finding Strongly Connected Components of a Digraph

Consider the digraph $D = (V(D), A(D))$.

1. Set up three arrays indexed by vertices of $V(D)$: an array of vertices called *father* and two arrays of integers called *order* and *low*. Also set up a boolean array called *used*, indexed by arcs of $A(D)$. Create an initially empty stack of vertices S. Set each element of *used* to be *false*, each element of *father* to be "undefined", and each element of *order* to be 0. Also set $i := 0$ and $v := s$ (s is the vertex we choose to start from).

2. Set $i := i + 1$, *order*$(v) := i$ and *low*$(v) := i$. Put v on S.

3. If there are no unused arcs outgoing from v then go to step 7.

4. Choose an unused arc $v \xrightarrow{a} u$. Set *used*(a) := *true*. If *order*$(u) = 0$ set *father*$(u) := v$, $v := u$ and then go to step 2.

5. If *order*$(u) >$ *order*(v) go straight back to step 3. Otherwise, if u is not on S (u and v do no belong to the same component) go to step 3.

6. (*order*$(u) <$ *order*(v) and both vertices are in the same component.) Set

$$\textit{low}(v) := \text{Min}(\textit{low}(v), \textit{order}(u))$$

   and go to step 3.

7. If *low*$(v) =$ *order*(v) then delete all vertices from S down to and including v; these vertices form a component.

8. If *father*(v) is defined then set

$$\begin{aligned} \textit{low}(\textit{father}(v)) &:= \text{Min}(\textit{low}(\textit{father}(v)), \textit{low}(v)) \\ v &:= \textit{father}(v) \end{aligned}$$

   and go to step 3.

9. (*father*(v) is undefined.) If there is a vertex u for which *order*$(u) = 0$ then let $v := u$ and go to step 2.

10. (All the vertices have been scanned.) Halt.

Selecting Arcs from a Digraph Lying on a Circuit

We may use the above technique to find all the arcs in a digraph which lie on a circuit. (This is required for the CSDD algorithm of Deadlock Checker). First we partition the vertices of the Digraph into strongly connected components, as described above. During the analysis a partition number $N(v)$ is assigned to each vertex v. We then scan through the arcs uv of the graph, removing any where $N(u) \neq N(v)$. It may be easily shown that those arcs which remain are exactly those which lie on a circuit in V.