cycle in the graph. An algorithm to detect a cycle in a graph requires an order of $n2$ operations, where n is the number of vertices in the graph.

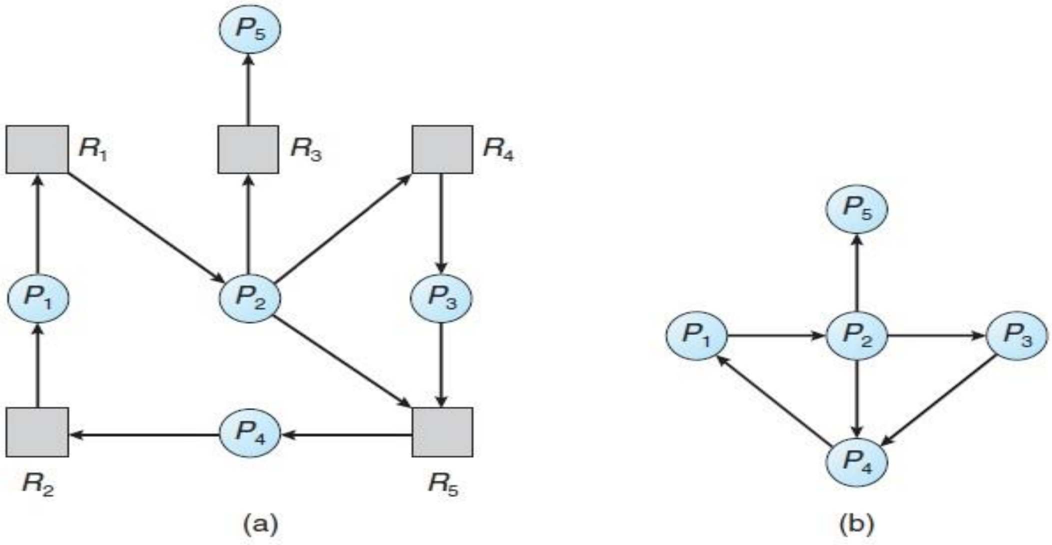

Figure 6-7 (a) Resource-allocation graph. (b) Corresponding wait-for graph

6.6.2. Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:

• **Available**. A vector of length m indicates the number of available resources of each type.

• **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

• **Request**. An $n \times m$ matrix indicates the current request of each process.

If ***Request***[i][j] equals k, then process Pi is requesting k more instances of resource type Rj .

The≤relation between two vectors is defined as in Banker's algorithm. To simplify notation, we again treat the rows in the matrices ***Allocation*** and ***Request*** as vectors; we refer to them as ***Allocation****i* and ***Request****i* **.** The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed. Compare this algorithm with the banker's algorithm.

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize ***Work*** = ***Available.*** For $i = 0, 1, ..., n-1$, if $\textbf{\textit{Allocation}}_i \neq 0$, then ***Finish***[*i*] = ***false.*** Otherwise, ***Finish***[*i*] = ***true.***

2. Find an index *i* such that both

a. ***Finish***[*i*] == ***false***

b. $\textbf{\textit{Request}}_i \leq \textbf{\textit{Work}}$

If no such *i* exists, go to step 4.

3. ***Work*** = ***Work*** + $\textbf{\textit{Allocation}}_i$

Finish[*i*] = ***true***

Go to step 2.

4. If ***Finish***[*i*] ==***false*** for some *i*, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if ***Finish***[*i*] == ***false,*** then process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state. You may wonder why we reclaim the resources of process P_i (in step 3) as soon as we determine that $\textbf{\textit{Request}}_i \leq \textbf{\textit{Work}}$ (in step 2b). We know that P_i is currently ***not*** involved in a deadlock (since $\textbf{\textit{Request}}_i \leq \textbf{\textit{Work}}$). Thus, we take an optimistic attitude and assume that P_i will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked. To illustrate this algorithm, we consider a system with five processes P_0 through P_4 and three resource types *A*, *B*, and *C*. Resource type *A* has seven instances, resource type *B* has two instances, and resource type *C* has six instances. Suppose that, at time T_0, we have the following resource-allocation state:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*P*0, *P*2, *P*3, *P*1, *P*4> results in ***Finish***[*i*] == *true* for all *i*.

Suppose now that process *P*2 makes one additional request for an instance of type *C*. The ***Request*** matrix is modified as follows:

	Request
	A B C
*P*0	0 0 0
*P*1	2 0 2
*P*2	0 0 1
*P*3	1 0 0
*P*4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process *P*0, the number of available resources is not sufficient to fulfil the requests of the other processes. Thus, a deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

6.7. Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system **recover** from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

6.7.1. Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes**. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked. Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term ***minimum cost*** is not a precise one. Many factors may affect which process is chosen, including:

1. What the priority of the process is

2. How long the process has computed and how much longer the process will compute before completing its designated task

3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)

4. How many more resources the process needs in order to complete

5. How many processes will need to be terminated

6. Whether the process is interactive or batch