CHAPTER 3

# Graph Representations

Graphs are deployed in many areas of parallel and distributed computing. Section 2.2 includes one example, where graphs are used for the representation of communication networks in parallel systems. Graphs are also employed for the representation of the task, communication, and dependence structure of programs. A first example is given in Figure 2.15, where the subtasks are drawn as ovals, and lines between the ovals represent the dependence relations between the respective tasks. This is the typical form of the pictorial representation of graphs (Berge [21]). Indeed, graphs are widely used for the representation of a parallel program's structure.

The task graph, often simply called DAG for directed acyclic graph, is the primary graph model used for program representation in task scheduling. Section 3.5 presents the task graph model and thoroughly discusses its properties and the concepts connected with it, as it is used heavily in the following chapters.

However, it is deemed essential to establish a broader background of graph models in order to have a general understanding of the task graph's bases, its relations to other major models, and its benefits as well as its drawbacks. Therefore, the next section will define basic graph concepts and Section 3.2 studies the graph as a model for program representation in general. It follows a discussion of the major graph models, namely, the dependence graph (Section 3.3), the flow graph (Section 3.4), and the task graph (Section 3.5).

## 3.1 BASIC GRAPH CONCEPTS

In order to discuss graph models, it is necessary to define graphs and establish some terminology. The initial three definitions of graph, path, and cycle are based on the notations given by Cormen et al. [42].

**Definition 3.1 (Graph)** *A graph $G$ is a pair $(\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ and $\mathbf{E}$ are finite sets. An element $v$ of $\mathbf{V}$ is called vertex and an element $e$ of $\mathbf{E}$ is called edge. An edge is a pair of vertices $(u, v)$, $u, v \in \mathbf{V}$, and by convention the notation $e_{uv}$ is used for an edge between the vertices $u$ and $v$.*

*In a* directed graph, *an edge $e_{uv}$ has a distinguished direction, from vertex $u$ to vertex $v$, hence $e_{uv} \neq e_{vu}$, and such an edge shall be referred to as a* directed edge. Self-loops—*edges from a vertex to itself* (*i.e.*, *$e_{uv}$ with $u = v$*)—*are possible. In an* undirected graph, *$e_{uv}$ and $e_{vu}$ are considered to be the same* undirected edge, *thus $e_{uv} = e_{vu}$, and since self-loops are forbidden $u \neq v$ for any edge $e_{uv}$.*

*It is said edge $e_{uv}$ is* incident on *the vertices $u$ and $v$, and if $e_{uv}$ is a directed edge, it is said that $e_{uv}$* leaves *vertex $u$ and* enters *vertex $v$. Similarly, if $e_{uv} \in \mathbf{E}$, vertex $v$ is* adjacent *to vertex $u$ and in an undirected graph, but not in a directed graph, vertex $u$ is adjacent to vertex $v$* (*i.e., in an undirected graph the adjacency relation is symmetric*). *The set $\{v \in \mathbf{V} : e_{uv} \in \mathbf{E}\}$ of all vertices $v$ adjacent to $u$ is denoted by* $\mathbf{adj}(u)$. *For the directed edge $e_{uv}$, $u$ is its* origin *vertex and $v$ its* destination *vertex.*

*The* degree *of a vertex is the number of edges incident on it. In a directed graph, it can be further distinguished between the* out-degree (*i.e., the number of edges leaving the vertex*) *and the* in-degree (*i.e., the number of edges entering the vertex*).

Figure 3.1 shows the pictorial representation of two graphs: Figure 3.1(*a*) an undirected graph and Figure 3.1(*b*) a directed graph. Vertices are represented by circles, undirected edges by lines, and directed edges by arrows. Both graphs are composed of the four vertices $u, v, w, x$. In the undirected graph, vertex $v$ has a degree of 2, since edges $e_{vx}$ and $e_{vw}$ are incident on it. Thus, vertices $x$ and $w$ are adjacent to $v$, and, as the graph is undirected, $v$ is also adjacent to them. In the directed graph, $v$ has a degree of 3 caused by the two entering edges $e_{uv}$ and $e_{xv}$ (in-degree $= 2$) and the leaving edge $e_{vw}$ (out-degree $= 1$). Vertex $x$ is adjacent to vertex $w$, caused by edge $e_{wx}$, whose origin is $w$ and whose destination is $x$. The edges $e_{xw}$ and $e_{wx}$ are identical in the undirected graph but are distinct in the directed graph. One of the edges leaving vertex $u$ also enters it and thereby builds the self-loop $e_{uu}$.

**Definition 3.2 (Path)** *A path $p$ in a graph $G = (\mathbf{V}, \mathbf{E})$ from a vertex $v_0$ to a vertex $v_k$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k\rangle$ of vertices such that they are connected by the edges $e_{v_{i-1}v_i} \in \mathbf{E}$, for $i = 1, 2, \ldots, k$. A path $p$* contains *the vertices $v_0, v_1, v_2, \ldots, v_k$ and the edges $e_{01}, e_{12}, e_{23}, \ldots, e_{(k-1)k}$ ($e_{ij}$ is short for $e_{v_iv_j}$) and the fact that a vertex $v_i$*

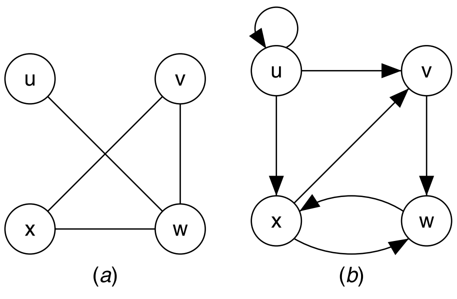


**Figure 3.1.** Pictorial representation of two sample graphs: (*a*) undirected graph and (*b*) directed graph. Both graphs consist of vertices $u, v, w, x$ and various edges; vertex $u$ has a self-loop (*b*).

*or an edge $e_{ij}$ is a member of the path p is denoted by $v_i \in p$ and $e_{ij} \in p$, respectively. Consequently, the path p from a vertex $v_0$ to a vertex $v_k$ is also defined by the sequence $\langle e_{01}, e_{12}, e_{23}, \dots, e_{(k-1)k} \rangle$ of edges. In this text, sometimes $p(v_0 \to v_k)$ is written to indicate the path goes from vertex $v_0$ to $v_k$. A path is* simple *if all vertices are distinct. The* length *of a path is the number of edges in the path. A* subpath *of a path $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$ is a contiguous subsequence of vertices $\langle v_i, v_{i+1}, \dots, v_j \rangle$ with $0 \le i \le j \le k$. Two paths $p_1 = \langle v_0, v_1, \dots, v_i \rangle$ and $p_2 = \langle u_0, u_1, \dots, u_j \rangle$ can be concatenated to build a new path $p = \langle v_0, v_1, \dots, v_i, u_1, \dots, u_j \rangle$, if $v_i = u_0$.*

In the sample undirected graph of Figure 3.1, the sequence $\langle u, w, x \rangle$ of vertices forms a simple path as does $\langle u, x, v, w \rangle$ in the directed graph, but the sequence $\langle w, x, v, w, u \rangle$ is a path in the undirected graph, which is not simple. Path $\langle u, x, v \rangle$ of length 2 in the directed graph is a subpath of the path $\langle u, x, v, w \rangle$ of length 3.

**Definition 3.3 (Cycle)** *A path $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a cycle if $v_0 = v_k$ and the path contains at least one edge. The cycle is* simple *if, in addition, the vertices $v_1, v_2, \dots, v_k$ are distinct.*

*The same cycle is formed by two paths $p = \langle v_0, v_1, \dots, v_{k-1}, v_0 \rangle$ and $p' = \langle v'_0, v'_1, \dots, v'_{k-1}, v'_0 \rangle$, if an integer r with $1 \le r \le k-1$ exists so that $v'_i = v_{(i+r) \bmod k}$ for $i = 0, 1, \dots, k-1$. A* subpath *of a cycle p is a contiguous subsequence of vertices of any of the paths, including p, forming the same cycle as p. A graph with no cycles is* acyclic.

An example for a simple cycle in Figure 3.1 is the path $\langle v, w, x, v \rangle$ in both sample graphs, while $\langle v, w, x, w, x, v \rangle$ establishes a cycle in the directed graph, which is not simple. The two paths $\langle v, w, x, v \rangle$ and $\langle w, x, v, w \rangle$ describe the same cycle ($r = 2$) and $\langle v, w \rangle$ is a subpath of this cycle.

The following lemma can be established about subpaths in cycles.

**Lemma 3.1 (Subpaths in Cycle)** *Let $p_c = \langle v_0, v_1, \dots, v_{k-1}, v_0 \rangle$ be a cycle in a directed graph $G = (\mathbf{V}, \mathbf{E})$. For any pair of vertices $v_i$ and $v_j$, $0 \le i < j \le k-1$, two subpaths $p(v_i \to v_j) = \langle v_i, v_{i+1}, \dots, v_{j-1}, v_j \rangle$ and $p(v_j \to v_i) = \langle v_j, v_{j+1}, \dots, v_{k-1}, v_0, \dots, v_{i-1}, v_i \rangle$ of the cycle exist. The path $p_r = \langle v'_0, v'_1, \dots, v'_{k-1}, v'_0 \rangle = \langle v_i, v_{i+1}, \dots, v_{j-1}, v_j, v_{j+1}, \dots, v_{k-1}, v_0, \dots, v_{i-1}, v_i \rangle$, concatenated from these two paths, forms the same cycle as $p_c$.*

*Proof*. The path $p_r$ exists and forms the same cycle as $p_c$ according to the definition of a cycle, since $v'_l = v_{(l+r) \bmod k}$ for $l = 0, 1, \dots, k-1$ with $r = i$. Since $p(v_i \to v_j)$ and $p(v_j \to v_i)$ are subpaths of the path $p_r$, they are by definition subpaths of the cycle $p_c$. □

**Definition 3.4 (Vertex Relationships)** *In a directed graph $G = (\mathbf{V}, \mathbf{E})$, the following relationships are defined. A vertex u is the* predecessor *of vertex v and correspondingly v is the* successor *of u, if and only if edge $e_{uv} \in \mathbf{E}$, $u, v \in V$. The vertex v is the successor of vertex u if it is adjacent to vertex u. The set $\{x \in \mathbf{V} : e_{xv} \in \mathbf{E}\}$*

*of all predecessors of $v$ is denoted by* $\mathbf{pred}(v)$ *and the set* $\{x \in \mathbf{V} : e_{vx} \in \mathbf{E}\}$ *of all successors of $v$, is denoted by* $\mathbf{succ}(v)$. *A vertex $w$ is called* ancestor *of vertex $v$ if there is a path $p(w \rightarrow v)$ from $w$ to $v$, and the set of all ancestors of $v$ is denoted by* $\mathbf{ance}(v) = \{x \in \mathbf{V} : \exists p(x \rightarrow v) \in G\}$. *Logically, a vertex $w$ is called* descendant *of vertex $v$ if there is a path $p(v \rightarrow w)$, and the set of all descendants of $v$ is denoted by* $\mathbf{desc}(v) = \{x \in \mathbf{V} : \exists p(v \rightarrow x) \in G\}$. *In the latter case it is sometimes said that vertex $w$ is* reachable *from $v$.*

Obviously, all predecessors are also ancestors and all successors are also descendants. Alternative notations are sometimes used when appropriate, for example, child or parent, given the analogy to the pedigree.

These are some examples from the directed graph in Figure 3.1($b$): vertex $x$ is a predecessor of $v$, which is a successor of $u$; hence, the set of predecessors of $v$ is $\mathbf{pred}(v) = \{u, x\}$. The set of successors of $w$ only comprises the single vertex $x$, $\mathbf{succ}(w) = \{x\}$. Finally, vertex $u$ is an ancestor of $w$ and $v$ is a descendent of $w$, but also its ancestor.

As seen in the examples above, a vertex relation is not necessarily unique in a cyclic graph.

**Lemma 3.2 (Vertex Relationships in Cycle)** *A vertex $v$ belonging to a cycle path $p_c$ is for any vertex of the cycle, $u \in p_c, u \neq v$, an ancestor and a descendant at the same time.*

*Proof*. The lemma follows directly from Lemma 3.1 and the notion of ancestor and descendant, since in a cycle $p_c$, for every two vertices $u$ and $v$, $u, v \in p_c, u \neq v$, there are two subpaths $p(u \rightarrow v)$ and $p(v \rightarrow u)$. □

Based on the notions of predecessors and successors, two vertex types can be further distinguished in a directed graph.

**Definition 3.5 (Source Vertex and Sink Vertex)** *In a directed graph $G = (\mathbf{V}, \mathbf{E})$, a vertex $v \in \mathbf{V}$ having no predecessors,* $\mathbf{pred}(v) = \emptyset$, *is named* source *vertex and a vertex $u \in \mathbf{V}$ having no successors,* $\mathbf{succ}(v) = \emptyset$, *is named* sink *vertex.*

Alternative notations for source vertex and sink vertex are entry vertex and exit vertex, respectively. The set of source vertices in a directed graph $G$ is denoted by $\mathbf{source}(G) = \{v \in \mathbf{V} : \mathbf{pred}(v) = \emptyset\}$ and the set of sink vertices by $\mathbf{sink}(G) = \{v \in \mathbf{V} : \mathbf{succ}(v) = \emptyset\}$.

### 3.1.1 Computer Representation of Graphs

For the complexity analysis of graph-based algorithms, in time and space, it is essential to consider the computer representation of graphs. There are two standard ways to represent a graph $G = (\mathbf{V}, \mathbf{E})$: as a collection of adjacency lists or as an adjacency matrix (Cormen et al. [42]).

| Vertex | List |
|---|---|
| u | →w |
| v | →x →w |
| w | →u →x →v |
| x | →v →w |

(*a*)

| Vertex | List |
|---|---|
| u | →u →v →x |
| v | →w |
| w | →x |
| x | →v →w |

(*b*)

**Figure 3.2.** The adjacency list representations of the two graphs in Figure 3.1: (*a*) for the undirected graph and (*b*) for the directed graph.

***Adjacency List Representation*** A graph can be represented as an array of $|\mathbf{V}|$ adjacency lists, one for each vertex in $\mathbf{V}$. The adjacency list belonging to vertex $u \in \mathbf{V}$ contains pointers to all vertices $v$ that are adjacent to $u$; hence, there is an edge $e_{uv} \in \mathbf{E}$. In other words, in vertex $u$'s adjacency list the elements of $\mathbf{adj}(u)$ are stored in arbitrary order. Figure 3.2 shows the adjacency list representations of the two sample graphs in Figure 3.1; in Figure 3.2(*a*) the one for the undirected graph and in Figure 3.2(*b*) the one for the directed graph.

For the directed graph, the sum of the lengths of the adjacency lists is $|\mathbf{E}|$, because for each edge $e_{uv}$ the destination vertex $v$ appears once in the list of vertex $u$. For an undirected graph every edge $e_{uv}$ appears twice, once in the list of $u$ and once in the list of $v$, due to the symmetry of the undirected edge; thus, the sum of the lengths of the adjacency list is $2|\mathbf{E}|$. Clearly, this representation form describes a graph $G$ completely, as there is a list for every vertex and at least one entry for every edge. The amount of memory required for a graph, directed or undirected, is consequently $O(\mathbf{V} + \mathbf{E})$.

In the previous asymptotic notation of the complexity, a common notational convention was adopted. The sign $|\,|$ for the cardinality (or size) of sets was omitted and it was written $O(\mathbf{V} + \mathbf{E})$ instead of $O(|\mathbf{V}| + |\mathbf{E}|)$. This shall be used in all asymptotic notations, but only there, since it makes them more readable and is nonambiguous.

The adjacency list representation has the disadvantage that there is no quicker way to determine if an edge $e_{uv}$ is part of a graph $G$ than to search in $u$'s adjacency list.

***Adjacency Matrix Representation*** The alternative representation of a graph as an adjacency matrix overcomes this shortcoming. A graph is represented by a $|\mathbf{V}| \times |\mathbf{V}|$ matrix $A$ and it is assumed that the vertices are indexed $1, 2, \ldots, |\mathbf{V}|$ in some arbitrary manner. Each element $a_{ij}$ of the matrix $A$ has one of two possible values: 1 if the edge $e_{ij} \in \mathbf{E}$ and 0 otherwise. Figure 3.3 depicts the two adjacency matrices for the graphs of Figure 3.1 with the vertices $u, v, w, x$ numbered $1, 2, 3, 4$, respectively.

Owing to the symmetry of undirected edges, the matrix of an undirected graph is symmetric along its main diagonal, which can be observed in the matrix of Figure 3.3(*a*). As the matrix is of size $|\mathbf{V}| \times |\mathbf{V}|$, the memory requirement of the adjacency matrix representation is $O(\mathbf{V}^2)$.

$$\begin{array}{c|c|c|c|c|}
 & u & v & w & x \\ \hline
u & 0 & 0 & 1 & 0 \\ \hline
v & 0 & 0 & 1 & 1 \\ \hline
w & 1 & 1 & 0 & 1 \\ \hline
x & 0 & 1 & 1 & 0 \\ \hline
\end{array}
\qquad
\begin{array}{c|c|c|c|c|}
 & u & v & w & x \\ \hline
u & 1 & 1 & 0 & 1 \\ \hline
v & 0 & 0 & 1 & 0 \\ \hline
w & 0 & 0 & 0 & 1 \\ \hline
x & 0 & 1 & 1 & 0 \\ \hline
\end{array}$$

(*a*) (*b*)

**Figure 3.3.** The adjacency matrix representations of the two graphs in Figure 3.1: (*a*) for the undirected graph and (*b*) for the directed graph.

For many algorithms the adjacency list is the preferred representation form, because it provides a compact way to represent sparse graphs—those for which $|\mathbf{E}|$ is much less than $|\mathbf{V}|^2$. For dense graphs, or when the fast determination of the existence of an edge is crucial, the adjacency matrix is preferred.

***Maximum Number of Edges*** From the above considerations it is easy to state the maximum number of edges in a graph.

*Directed Graph* The maximum number of edges in a directed graph $G = (\mathbf{V}, \mathbf{E})$ is $|\mathbf{V}|^2$, that is, the number of elements in the adjacency matrix, as every vertex may have an edge to every other vertex including itself.

$$|\mathbf{E}| \leq |\mathbf{V}|^2. \tag{3.1}$$

*Undirected Graph* The symmetry of an undirected edge reduces the maximum number of edges by more than half, compared to a directed graph, as already shown by the symmetric adjacency matrix. The maximum number of edges of an undirected graph $G = (\mathbf{V}, \mathbf{E})$ is limited by

$$|\mathbf{E}| \leq \sum_{i=1}^{|\mathbf{V}|-1} i = \tfrac{1}{2}|\mathbf{V}|(|\mathbf{V}| - 1). \tag{3.2}$$

The maximum number of edges in a directed *acyclic* graph is identical to that in an undirected graph. It follows that in both graph forms—directed and undirected—the number of edges is $O(\mathbf{V}^2)$. In describing the running time of a graph-based algorithm in the following, the size of the input shall be measured in terms of the number of vertices $|\mathbf{V}|$ and the number of edges $|\mathbf{E}|$. Substituting the size of $\mathbf{E}$ by $O(\mathbf{V}^2)$ would be too rough an approximation, as the number of edges of a graph varies largely among graphs.

### 3.1.2 Elementary Graph Algorithms

This section concludes by reviewing some of the well-known elementary graph algorithms, which built the foundations of many others (Cormen et al. [42]).

Two simple search algorithms, complementary in their approach, can be considered the most important graph algorithms, the *breadth first search* (BFS) and the *depth first search* (DFS). Both can be applied to a directed or undirected graph $G = (\mathbf{V}, \mathbf{E})$.

***BFS* (*Breadth First Search*)** The BFS searches a graph $G$ beginning with a specified start vertex $s$. It examines all vertices adjacent to $s$ and then continues with the vertices adjacent to these vertices and so on until all vertices reachable from $s$ have been considered. Breadth first search is so named because it first discovers the vertices at distance 1 of $s$, then at distance 2, and so on. The *distance* between two vertices $u$ and $v$ is defined as the minimum number of edges that must be traversed to reach $v$ from $u$; or expressed in another way, distance is the length of the *shortest path* from $u$ to $v$. When during the BFS the predecessor, from which a vertex is discovered, is stored, upon termination a shortest path between $s$ and any vertex $v$ reachable from $s$ is given by the recursive list of predecessors of $v$. Algorithm 1 briefly outlines BFS, which uses a FIFO (first in first out) queue as its main data structure.

***Algorithm 1 BFS(G, s)***

```
Put $s$ into a FIFO queue $Q$; mark $s$ as discovered
while $Q \neq \emptyset$ do
  Let $u$ be the first vertex of $Q$; remove $u$ from $Q$
  for each $v \in \mathbf{adj}(u)$ do
    if $v$ not discovered then
      Put $v$ into $Q$ and mark $v$ as discovered
    end if
  end for
end while
```

***DFS* (*Depth First Search*)** The DFS searches a graph $G$ using the depth first approach. It moves deeper into the graph before finishing the examination of all vertices adjacent to a vertex $u$. Only when all vertices reachable from a vertex $v$ adjacent to $u$ have been examined, does DFS return to examine the remaining undiscovered vertices adjacent to $u$. Consequently, DFS is readily expressed as a recursive algorithm, using a subroutine, here named DFS-Visit, which is recursively called for any undiscovered adjacent vertex as soon as it is found. DFS-Visit searches the subgraph spanned by all reachable vertices from its parameter vertex $u$ and their adjacent edges. Algorithm 3 outlines the DFS-Visit subroutine and Algorithm 2 shows the main routine of DFS, which calls DFS-Visit for every undiscovered vertex $v \in G$.

Both the BFS and the DFS have a runtime complexity of $O(\mathbf{V} + \mathbf{E})$, since for each vertex ($O(\mathbf{V})$), every adjacent vertex is examined once and the total number of

***Algorithm 2 DFS(G)***

```
for each $v \in \mathbf{V}$ do
  if $v$ not discovered then
    DFS-Visit($v$)
  end if
end for
```

***Algorithm 3 DFS-Visit(u)***

```
for each $v \in \mathbf{adj}(u)$ do
  if $v$ not discovered then
    Mark $v$ as discovered
    DFS-Visit($v$)
  end if
end for
Mark $u$ as finished
```

adjacent vertices is $O(\mathbf{E})$ (see also Section 3.1.1). For more details and an in-depth analysis of the properties of the two algorithms please refer to Cormen et al. [42].

***Topological Order*** Now an important concept for directed acyclic graphs is considered—the topological order of their vertices (Cormen et al. [42]). Directed acyclic graphs build an essential class of graphs for task scheduling, because they are utilized for the representation of programs (Section 3.5) in scheduling algorithms. The topological order is defined as follows.

**Definition 3.6 (Topological Order)** *A topological order of a directed acyclic graph $G = (\mathbf{P}, \mathbf{E})$ is a linear ordering of all its vertices such that if $\mathbf{E}$ contains an edge $e_{uv}$, then $u$ appears before $v$ in the ordering.*

To illustrate Definition 3.6, consider the small directed acyclic graph in Figure 3.4. The topological order of the graph's vertices can be interpreted as a horizontal arrangement of the vertices (i.e., a linear order), in such a way that all edges are directed from left to right. This arrangement of the graph in Figure 3.4 is depicted in Figure 3.5.

The acyclic property is crucial for the topological order; otherwise no such order exists.

**Lemma 3.3 (Topological Order and Directed Graphs)** *A directed graph $G = (\mathbf{P}, \mathbf{E})$ is acyclic if and only if there exists a topological order of its vertices.*

*Proof.* $\Rightarrow$: Suppose no topological order exists for $G$. Thus, for any ordering of the vertices of $G$, there is at least one edge $e_{vu}$ with $u$ appearing before $v$ in the list. Consequently, there must be a path $p(u \to v)$ from $u$ to $v$; otherwise $v$ and all of its ancestors that lie between $u$ and $v$ in the ordering could be inserted just before $u$,

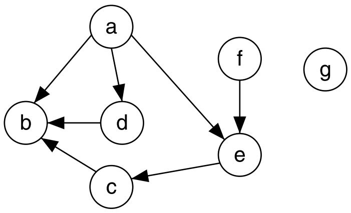


**Figure 3.4.** A directed acyclic graph.

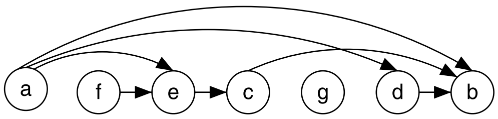


**Figure 3.5.** The directed acyclic graph of Figure 3.4 arranged in topological order; note that all directed edges go from left to right.

making the edge $e_{vu}$ comply with the topology order without making a new edge violating it. With the path $p(u \rightarrow v)$, however, $G$ is cyclic, since edge $e_{vu}$ builds a cycle together with the path $p(u \rightarrow v)$.

$\Leftarrow$: Suppose $G$ is cyclic; hence, it has at least one simple cycle $p_c = \langle v_0, v_1, \ldots, v_{k-1}, v_0 \rangle$. Consider the distinct vertices $v_0, v_1, \ldots, v_{k-1}$ of $p_c$. In a topological order, vertex $v_i$ must appear before $v_{i+1}$, $0 \leq i < k-1$, imposed by the edge $e_{i,i+1} \in p_c$. That implies $v_0$ comes before $v_{k-1}$, but then edge $e_{k-1,0}$ does not comply with the condition of the topological order. Consequently, no topology order exists for the vertices of $p_c$. Since no topology order can be found for the vertices of $p_c$, no topology order exists for $G$. $\square$

The vertices of a directed acyclic graph can be sorted into topological order by a simple DFS-based algorithm, which is outlined in Algorithm 4 (Cormen et al. [42]). As soon as a vertex is marked finished in the DFS (see DFS-Visit, Algorithm 3), it is inserted onto the front of a list and upon termination of DFS, the list holds the topologically ordered vertices.

***Algorithm 4 Topological-Sort(G)***

```
Execute DFS($G$) with the following addition:
  Insert each vertex of $G$ onto the front of a list $L$ as soon as it is marked finished
Return $L$
```

The correctness of Algorithm 4 can be verified by the following considerations. It is ensured by the main part of DFS (Algorithm 2) that every vertex is discovered; therefore, every vertex is eventually marked as finished and inserted onto the front of the list. When a vertex $u$ is marked as finished, none of the vertices already in the list can have an edge to $u$. If there were such a vertex $v$ in the list $L$ with $e_{vu} \in \mathbf{E}$, $u$ would be adjacent to $v$ and DFS-Visit would have been recursively called for $u$ before $v$ was marked finished. That is, $u$ would have finished before $v$, which is a contradiction.

Given that Topological-Sort($G$) is based on DFS, its runtime complexity is $O(\mathbf{V} + \mathbf{E})$.

## 3.2 GRAPH AS A PROGRAM MODEL

Sinnen and Sousa [173] classify some of the well-known graph theoretic models for the representation of parallel computations. They extracted several characteristics that are shared among most of the graph models. Premised on their findings, a general graph model is defined and its properties are analyzed in the next paragraphs.

**Definition 3.7 (Graph Model)** *In a graph theoretic abstraction, a program consists of two kinds of activity—computation and communication. The computation is associated with the vertices of a graph and the communication with its edges. A vertex is called* node *and the computation associated with it* task. *A task can range from an atomic instruction/operation (i.e., an instruction that cannot be divided into smaller instructions) to threads or compound statements such as loops, basic blocks, and sequences of these. All instructions or operations of one task are executed in sequential order; there is no parallelism within a task. A node is at any time involved in only one activity—computation or communication.*

Note that the granularity of the general graph model is not restricted in any way, leaving this to the individual graph models considered in the next sections. For instance, in the program examples of Section 2.5, one simple mathematical operation (e.g., summation or multiplication) is considered a task. An essential property of the nodes is their strictness.

**Definition 3.8 (Node Strictness)** *The nodes are strict with respect to both their inputs and their outputs: that is, a node cannot begin execution until all its inputs have arrived, and no output is available until the computation has finished and at that time all outputs are available for communication simultaneously.*

***Branching*** None of the graph models covered in this chapter incorporates a branching concept. In other words, there is no node type deciding which of the successors participates in the program execution. An example is a branch node that represents an `if...else` statement, shown in Figure 3.6: the `if` node has two edges—one representing the *true* and the other the *false* branch.

In any execution of the program represented by the entire graph, the `if` node only leads to the subsequent execution of one of the two subgraphs spanned by the branches. This concept is used in control flow graphs (CFGs) (Wolfe [204]) for modeling control dependence and its flow. Not allowing branching has two consequences: (1) all nodes

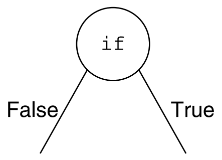


**Figure 3.6.** An `if...else` statement represented as a branch node.

of a graph participate in the execution of the program; and (2) conditional statements cannot directly be reflected by the graph structure. Thus, either the control structure of a program must be encapsulated within a node, which potentially leads to larger nodes (see granularity in Section 2.4.1), or it must be transformed into data dependence structure as addressed in Section 2.5.3.

A graph adhering to the above definitions reflects with its node and edge structure a program's decomposition into tasks and their communication structure. As mentioned earlier, such a graph is not capable of representing the control dependence, but only the data dependence structure. In contrast, data dependence is very well described by the structure of such a graph. One of the most prominent graph models is the dependence graph that exposes all types of data dependence, studied in Section 2.5.1.

### 3.2.1 Computation and Communication Costs

Many utilization areas of program graph models require knowledge about the computation and communication costs of the nodes and edges, respectively, unless it can be assumed they are uniform. Commonly, these costs are measured in time—the time a computation or communication takes on the respective target system—and are incorporated into the graph model by assigning weights to the graph elements. In terms of the general graph model of Definition 3.7 this can be defined as follows.

**Definition 3.9 (Computation and Communication Costs)** *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a graph representing a program $\mathcal{P}$ according to Definition 3.7. The nonnegative weight $w(n)$ associated with node $n \in \mathbf{V}$ represents its computation cost and the nonnegative weight $c(e)$ associated with edge $e \in \mathbf{E}$ represents its communication cost.*

Examples for graph models employing such weights are some flow graphs (Section 3.4) and the task graph (Section 3.5). Further weights might be associated with the graph elements; for example, in the flow graph each edge has a weight representing its delay.

### 3.2.2 Comparison Criteria

Having defined a common principle for graph models implies that there are characteristics that allow one to differentiate them. Sinnen and Sousa [174] present several criteria for analyzing and distinguishing the various models. These criteria are based on similar aspects as considered during the discussion of subtask decomposition in Section 2.4. This is not surprising given the fact that the creation of a graph involves the decomposition of the program into tasks and the knowledge of their communication relations. The three main criteria are:

- *Computation Type.* What type of computation can be efficiently represented with the graph model? Graph models can be distinguished regarding the modeled granularity and the ability to reflect iterative computations and regularity.
- *Parallel Architectures.* For which parallel system architecture is the graph model usually employed? As mentioned in Chapter 2, generic parallel systems can be classified according to their instruction and data stream type, their memory architecture, and their programming model.
- *Algorithms and Techniques.* What are the algorithms and techniques that can be applied to the graph model? Typically, graphs are used for dependence analysis, program transformations, mapping, and scheduling.

Apart from these main topics, practical issues like the computer representation and the size of the graphs are also discussed by Sinnen and Sousa [174].

During the following review of the individual graph models, these criteria will be helpful when explaining the motivation for the respective graph model and discussing its distinctive features. The first graph to be examined is the dependence graph, which is intuitively derived from the dependence analysis in Section 2.5.

## 3.3 DEPENDENCE GRAPH (DG)

Recall the discussion of dependence in Section 2.5, where directed dependence relations are analyzed between tasks. Intuitively, one can represent the programs' tasks as nodes and the dependence relations as edges, corresponding to the graph model defined in the previous section. Consider the program in Example 1 of Section 2.5, which contains flow dependence relations. Representing every task, in this case every statement, by a node and every dependence relation by an edge directed from the cause to the dependent task, a graph reflecting the dependence structure is obtained. Figure 3.7 repeats the program code of Example 1 and depicts the corresponding dependence graph. From this graph one intuitively perceives that tasks 2 and 3 can be executed concurrently.

Next, the dependence graph is defined formally.

**Definition 3.10 (Dependence Graph (DG))** *A dependence graph is a directed graph $DG = (\mathbf{V}, \mathbf{E})$ representing a program $\mathcal{P}$ according to the graph model of Definition 3.7. The nodes in $\mathbf{V}$ represent the tasks of $\mathcal{P}$ and the edges in $\mathbf{E}$ the dependence relations between the tasks. An edge $e_{ij} \in \mathbf{E}$ from node $n_i$ to $n_j$, $n_i, n_j \in \mathbf{V}$, represents the dependence of node $n_j$ on node $n_i$.*

```
1: a = 2
2: u = a + 2
3: v = a * 7
4: x = u + v
```

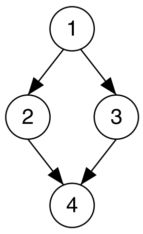


**Figure 3.7.** The code from Example 1 and the corresponding dependence graph.

A dependence graph *DG* reflects the dependence structure of a program $\mathcal{P}$, which imposes a partial order of the nodes (i.e., the tasks) in an execution. Every edge $e_{ij} \in DG$ constrains the precedence of $n_i$ over $n_j$ in the execution order of the nodes. Graphically, one can imagine that the nodes have to be executed following the edges—the execution of a program progresses with the edge direction, never against it. Note that the node strictness (Definition 3.8) accurately reflects the nature of dependence: a node can only start if all of its entering dependence relations are fulfilled.

**Definition 3.11 (Feasible Program)** *A program $\mathcal{P}$ is feasible if and only if a task order can be found that complies with the precedence constraints of the dependence relations.*

Clearly, the feasibility of a computation is indicated by its dependence graph.

**Lemma 3.4 (DG of Feasible Program Is Acyclic)** *A program $\mathcal{P}$ is feasible if and only if its dependence graph DG is acyclic.*

*Proof*. This lemma can easily be proved by contradiction using Lemma 3.3 (topological order and directed graphs).

$\Rightarrow$: Suppose *DG* is cyclic. According to Lemma 3.3, no topological order exists for the nodes of a cyclic graph. But then, for any order of the nodes, at least two nodes are not in precedence order. As the nodes represent the tasks of $\mathcal{P}$, also no precedence order exists for them—$\mathcal{P}$ is not feasible.

$\Leftarrow$: Suppose $\mathcal{P}$ is not feasible, thus no order of the tasks can be found that complies with the precedence constraints. Consequently, no topological order exists for *DG*. Then, however, *DG* is cyclic according to Lemma 3.3. □

According to the graph model of Definition 3.7, the communication of a program is represented by the edges of a graph. A further level of abstraction is introduced by the DG, as only the dependence relations caused by communication are considered. From this it is clear that the dependence reflected by a DG is the data dependence of a program. As discussed in Section 2.5.1, there are three types of data dependence: flow dependence, antidependence, and output dependence. All these types can be represented and distinguished by the DG, albeit often the mere fact that a dependence

```
1: a = 2
2: v = a * 5
3: u = v + 2
4: v = a * 7
5: x = u + v
```

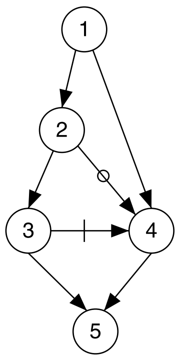


**Figure 3.8.** The code from Example 2 and the corresponding dependence graph. A line through the arrow of an edge denotes antidependence and a circle through the arrow denotes output dependence; a plain arrow stands for flow dependence.

exists is relevant. For distinction in a pictorial representation, the edge of an antidependence is sometimes drawn as an arrow with a line crossing through and the edge of an output dependence as an arrow with a circle through it (Banerjee [15], Wolfe [204]). A flow dependence edge is denoted by a plain arrow. Figure 3.8 shows a DG that uses this kind of notation for the program of Example 2: task 4 is antidependent on task 3 and also output dependent on task 2. All other dependence relations are flow, or real, dependences.

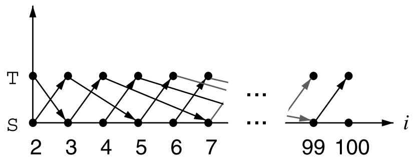
## 3.3.1 Iteration Dependence Graph

A typical utilization area for dependence graphs is in compilers. The DG gives a compiler a way to capture the precedence constraints that prevent it from reordering operations in the program. Parallelizing compilers usually focus on the parallelization of loops, as they commonly accommodate the largest share of computational load. A logical specialization of the DG is therefore the iteration dependence graph, reflecting dependence relations in loops. The theoretical background for the transition from a simple DG to an iteration dependence graph is given in Section 2.5.2, where dependence relations in loops were analyzed. Reconsider the loop in Example 5, here shown in Figure 3.9(*a*). A graphical representation of the dependence relations—the iteration dependence graph—is given in Figure 3.9(*b*).

Each instance of the statements `S` and `T` is modeled as a task and thus a node of the graph, while the edges represent the dependence relations between these instances. In Figure 3.9(*b*), a node is drawn as a dot in the coordinate system spanned by the statements and the iterations of the loop. The dependence distance vector corresponds to the spatial vector drawn in the graph illustration, whereby the vertical dimension is for the distinction between the tasks. The short arrows going bottom–up reflect the uniform dependence and the arrows going top–down reflect the nonuniform dependence between instances of the tasks `S` and `T` (see also Section 2.5.2). Due to the nonuniform dependence between instances of tasks `S` and `T`, the graph is irregular.

```
for i = 2 to 100 do
  S: A(i) = B(i+2) + 7
  T: B(i*2+1) = A(i-1) + C(i+1)
end for
```

(*a*)



(*b*)

**Figure 3.9.** (*a*) The loop from Example 5 and (*b*) the corresponding iteration dependence graph.

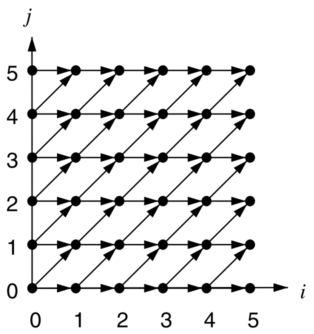
In a loop nest, the iteration DG gains one dimension for every loop. Figure 3.10(*a*) displays the double loop from Example 7 and its iteration DG. Here, the approach differs from the one in Figure 3.9, since each node of the graph encapsulates all statements of one iteration: that is, the entire loop kernel is one task. A more precise decomposition of the loop body into a task for every statement, as done with the iteration DG in Figure 3.9, would add one more dimension to the graph. It depends on the context in which the graph is generated if this is necessary and/or desirable. Note that if the entire loop body is modeled as one task, the graph is only able to reflect interiteration dependence. Certain parallelization techniques, for example, software pipelining (e.g., Aiken and Nicolau [8]), are based on both intra- and interiteration dependence.

In Section 2.5.2, the distance vectors $(1, 1)$ and $(1, 0)$ were identified for the dependence relations in the code of Figure 3.10(*a*): S$(i + 1, j + 1)$ depends on T$(i, j)$, and T$(i + 1, j)$ depends on S$(i, j)$. As with the single loop of Figure 3.9(*a*), the arrows in the iteration DG correspond directly to the dependence distance vectors; but recall that the two statements of the kernel are merged into one task, that is, one node in the graph. The uniform dependence relations of the loop create a regular iteration DG.

For the construction of iteration dependence graphs, it is obvious that knowledge of the distance vectors of all dependence relations is crucial. In other words, if the distance vector cannot be determined for every single dependence relation, the complete

```
for i = 0 to 5 do
   for j = 0 to 5 do
      S: A(i+1,j) = B(i,j) + C(i,j)
      T: B(i+1,j+1) = A(i,j) + 1
   end for
end for
```

(*a*)



(*b*)

**Figure 3.10.** (*a*) The double loop from Example 7 and (*b*) the corresponding iteration dependence graph.

iteration DG cannot be constructed. This fact is noteworthy, especially in conjunction with the observation made at the end of Section 2.5.2: the known techniques of dependence analysis can only compute—in reasonable time—the distance vectors for dependence relations based on simple, albeit common, types of subscript functions. Not to forget that it is sometimes even impossible to determine the dependence relations, for example, when array subscripts are functions of input variables of the program. The only solution is then to create a dependence edge for every possible dependence relation in order to obtain a conservative dependence graph, that is, a graph that includes all edges that the true dependence graph would have.

### 3.3.2 Summary

As the name indicates, the dependence graph is a representation of the dependence structure of a program. In its general form, as in Definition 3.10, the DG is not bound to any particular computation type or granularity. This general theoretical form is widely employed, whenever it is necessary to reason about the dependence structure of a program, the feasibility of computations, or the validity of program transformations (see the literature referenced in Section 2.5.1). In practical usage, the DG is limited to a coarse grained or partial representation of a program, because its size (and with it its computer representation) grows with the number of tasks.

The dependence structure of cyclic computations is represented by the iteration dependence graph, where every node of the graph is associated with coordinates of the iteration space spanned by the index variables of the loop nest. For dependence analysis of loops and, above all, for loop transformations, the iteration DG is a valuable instrument. Banerjee et al. [17] survey many of the existing techniques. Although several of them do not require the explicit construction of the graph, the underlying theory of these techniques is based on the dependence graph. Granularity in loops is typically small to medium, which is therefore the common granularity of iteration dependence graphs. The size issue of the general DG is evaded by benefitting from the regular iterative structure of the computation. A compact computer representation, given that the dependence distances are uniform, comprises only the tasks of the loop kernel, the intraiteration dependence relations, and the distance vectors of the interiteration dependence relations.

Iteration DGs are employed not only in parallelizing compilers but also in VLSI array processor design (Kung [109]). There, the iteration DG, constructed from an initial application specification—typically signal processing—serves as an intermediate representation for the mapping and scheduling of the application to array processors. Those graphs possess two distinctive attributes: (1) a node represents one iteration, not only a part of it; and (2) the dependence relations are local, that is, the absolute values of the distance vectors' elements are typically 1 or 0 (Karp et al. [101]). The mapping and scheduling process transforms the iteration DG into a flow graph, which is treated in the next section.

While the dependence graph has no affinity for any particular parallel architecture, the iteration dependence graph, owing to its regular iterative structure, is usually employed for homogeneous systems, with SIMD or SPMD streams.

## 3.4 FLOW GRAPH (FG)

The iteration dependence graph, as discussed in the previous section, is not the only graph model for the representation of cyclic computations. Intuitively, a model for such computations can benefit from the inherent regular structure, as insinuated during the discussion of the computer representation of an iteration dependence graph in the previous section. It seems to be sufficient for a model to reflect the tasks of one iteration with their intra- and interiteration communications.[1]

The flow graph (FG) achieves a concise representation of the structure of an iterative computation by introducing timing information into the graph, which allows one to distinguish between intra- and interiteration communication. This is illustrated with Example 10.

***Example 10 Flow Graph***

```
for i = 1 to N do
  S: A(i) = C(i) * D(i+2)
  T: B(i+1) = A(i) + 10
  U: C(i+2) = A(i) + B(i)
end for
```

Analyzing the data dependence of its loop (Section 2.5.2), the following communications can be identified. Considering the three statements S, T, and U as tasks, there are two intraiteration communications—the output of S is the input of T and U, caused by $A(i)$—and two interiteration communications—the output of T$(i)$ is the input of U$(i + 1)$ (array $B$) and the output of U$(i)$ is the input of S$(i + 2)$ (array $C$). The flow graph for this computation is then as depicted in Figure 3.11.

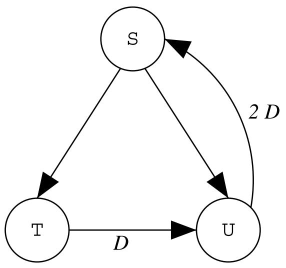


**Figure 3.11.** Flow graph of code in Example 10.

[1]Due to the close connection between dependence and communication, many concepts of dependence can be directly applied to the treatment of communication. Therefore, those concepts will be employed when appropriate for communication without further formalization, unless it is deemed ambiguous.

The graph consists of three nodes for the tasks S, T, and U, and four edges for the communications between these tasks. The edges, which reflect the communications between nodes, are drawn as arrows as usual, whereby an interiteration communication is distinguished by a label showing its communication distance, that is, dependence distance (see Section 2.5.2). As is often done in the literature (e.g., Parhi [145], Yang and Fu [208]), the communication distance, which is the delay of the communication in terms of iterations, is expressed as multiples of $D$, where $D$ stands for one iteration. So edge $e_{\mathrm{TU}}$ has a delay of $D$ for the communication distance of one iteration, and edge $e_{\mathrm{US}}$ has a delay of $2D$, that is, two iterations. Before examination of this graph model's properties, it is defined formally.

**Definition 3.12 (Flow Graph)** *A flow graph is a directed graph $FG = (\mathbf{V}, \mathbf{E}, D)$ representing a program $\mathcal{P}$ of an iterative computation according to the graph model of Definition 3.7. The nodes in $\mathbf{V}$ represent the tasks of $\mathcal{P}$ and the edges in $\mathbf{E}$ the communications between the tasks. An edge $e_{ij} \in \mathbf{E}$ from node $n_i$ to $n_j$, $n_i, n_j \in \mathbf{V}$, represents a communication from node $n_i$ to node $n_j$. Each edge $e \in \mathbf{E}$ is associated with a nonnegative integer weight $D(e)$, representing a delay count.*

During execution of the program $\mathcal{P}$, every task represented by the nodes of $FG$ is executed once in each iteration. Only when all nodes have finished their execution, can a new iteration begin. In distinction from the data-driven execution model outlined later, this form of execution is called iteration-driven. No communications or dependences exist in $\mathcal{P}$ other than those reflected by the edges of $FG$. The delay $D(e)$ associated with each edge reflects the number of iterations a communication is delayed between its output from the origin node until its input into the destination node. Communication between nodes within one iteration (i.e., intraiteration communication) has consequently the delay value 0. In the sample graph of Figure 3.11, the edges $e_{\mathrm{SU}}$ and $e_{\mathrm{ST}}$ have zero delay, which by convention is omitted in the illustration.

It is possible for a flow graph to have parallel edges, that is, distinct edges that have the same origin and destination node. Graphs with this property are sometimes called multigraphs (Cormen et al. [42]). Parallel edges in the flow graph must differ, however, on their delay value, by which they can thus be distinguished. Figure 3.12 exhibits an example program whose flow graph contains parallel edges. Two communications go from task S to task T: one intraiteration (via $A(i)$) and one interiteration (via $A(i-1)$) with a delay of 1.

```
for i = 1 to N do
  S: A(i) = C(i) * D(i)
  T: B(i) = A(i) + A(i-1)
end for
```

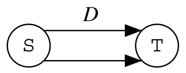


**Figure 3.12.** An iterative computation whose flow graph has parallel edges.

The flow graph *FG* does not contain information about the number of iterations. This also means that the number of iterations does not have to be known when constructing the graph, as is the case, for instance, with a loop whose index bound is a variable (*N* in Example 10).

It should be mentioned, that all communications must be uniform, in order to represent their communication distance as one integer value. Nonuniform communication, as found in the loop of Figure 3.9, cannot be expressed as one integer value. Yet, a flow graph can still be constructed in such cases using a conservative approximation of the distance, for example, "$+1$" to denote an unknown dependence distance greater than or equal to one. For certain loop parallelization algorithms, such information can be sufficient. If the accurate representation of the dependence structure is indispensable, a flow graph can only be used with uniform communication. This establishes an important limitation of the flow graph, which relates to the representable computation types.

In comparison to the DG or the iteration DG (Section 3.3), the flow graph has two essential characteristics. First, nodes in the flow graph represent tasks that can be executed several times, and not instances of tasks as in the DG, where each represented node is executed exactly once. Second, the flow graph is, in general, not acyclic. Cycles can arise in connection with interiteration dependence, but in contrast to the DG, the represented computation is still feasible.

When reading the code of Example 10, one concludes that it is a feasible program, but in fact its FG in Figure 3.11 contains two cycles—$\langle \mathtt{S}, \mathtt{U}, \mathtt{S} \rangle$ and $\langle \mathtt{S}, \mathtt{T}, \mathtt{U}, \mathtt{S} \rangle$. The flow graph breaks a limitation imposed on the dependence graph: in contrast to the DG, a flow graph can include cycles. As expressed by Lemma 3.4, a DG must be acyclic in order to represent a feasible computation. So, how can a flow graph, based on the same general graph model of Definition 3.7, be a valid representation of a feasible program? According to Definition 3.8 (node strictness), the nodes must be strict regarding their input and output. Even though an FG models the data flow among nodes, and not as the DG the dependence relations, communications among nodes create (flow) data dependence relations (see Section 2.5.1). In other words, the edges implicitly represent dependence relations; thus, a communication cycle would lead to a dependence cycle.

The reason an FG remains feasible, despite the cycles, lies in the introduction of delays on the edges, which prevent cycles in the graph from turning into dependence cycles. In the flow graph, every cycle must contain at least one delayed edge, breaking the dependence chain. Seen the other way around, paths in flow graphs are only closed to cycles by interiteration communications, which by definition are delayed.

**Lemma 3.5 (Feasible Flow Graph)** *A flow graph $FG = (\mathbf{V}, \mathbf{E}, D)$ represents a feasible iterative computation $\mathcal{P}$ if and only if any cycle $p_c$ in FG contains at least one edge e with a nonzero delay $D(e) \neq 0$:*

$$\forall p_c \in FG \, \exists e \in \mathbf{E} : e \in p_c \wedge D(e) \neq 0. \tag{3.3}$$

*Proof*. $\Rightarrow$: Suppose *FG* contains at least one cycle whose edges all have zero delay. This corresponds to a cycle in the *DG*, which, however, must be acyclic to represent a feasible program (Lemma 3.4).

$\Leftarrow$: The program's feasibility is shown by demonstrating that the *DG* corresponding to the flow graph has no cycles (Lemma 3.4). One node in *DG* represents one *instance* of a node in *FG*; that is, *DG* consists of *i*-times the nodes of *FG*, therefore $|\mathbf{V}_{DG}| = i \times |\mathbf{V}_{FG}|$, with $\mathbf{V}_{DG}$ and $\mathbf{V}_{FG}$ being the node sets of *DG* and *FG*, respectively, and $i$ the number of iterations. (Figure 3.13 shows the *DG* of the flow graph in Figure 3.11 assuming three iterations—$N = 3$ in the underlying program of Example 10—and each iteration comprises the nodes `S`, `T`, and `U`.) Communication edges in *FG* with zero delay $\{e \in \mathbf{E}_{FG} : D(e) = 0\}$, correspond to dependence edges in *DG* between the nodes of the same iteration; thus, there are $i \times |\{e \in \mathbf{E}_{FG} : D(e) = 0\}|$ edges of this type. (These are three times the edges $e_{\mathtt{SU}}$ and $e_{\mathtt{ST}}$ in Figure 3.13.) As every cycle in *FG* contains at least one nonzero edge, there cannot be a cycle in *DG* among the nodes of one iteration. Edges with nonzero delay are directed from an earlier to a later iteration, never the other way around, because the weight is nonnegative, $D(e) \geq 0 \forall e \in \mathbf{V}_{FG}$. (The graph of Figure 3.13 has three such edges: $e_{\mathtt{T(1)U(2)}}$, $e_{\mathtt{T(2)U(3)}}$, and $e_{\mathtt{U(1)S(3)}}$.) This means that the entering edges of a node always have their origin nodes in the same or a previous iteration, while leaving edges have their destination node in the same or a subsequent iteration. But then there is also no dependence cycle in *DG* across iterations, as no path can return to its origin node. $\square$

The technique employed in the above proof—creating a dependence graph from a flow graph—is called unrolling or unfolding (Parhi [144]). With the inverse technique, called projection, a flow graph can be obtained from an iteration DG (Kung [109]). Both techniques, which show the close relation of the graph models, will be considered in the following section.

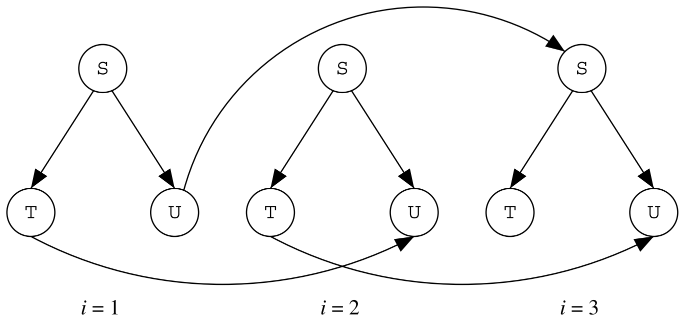


**Figure 3.13.** Dependence graph, created by unrolling the flow graph of Figure 3.11 for three iterations ($N = 3$ in Example 10).

### 3.4.1 Data-Driven Execution Model

A communication edge of a flow graph can also be interpreted as a communication between two nodes via an intermediate queue. The communication data is written by the origin node of the communication edge into the queue from where it is read by the destination node. A delay arises when other data items, often designated tokens (Kung [109]), already populate the queue at the time a new item is placed in the queue, since these items are read before the new one, due to the FIFO (first in first out) characteristic of the queue. When the execution of the flow graph starts, the queues of the edges with a nonzero delay contain initial data items, or tokens, whose number is given by the delay count $D(e)$ of the edges. For the code of Example 10, these initial data items correspond to the initial values of the array elements of $B$ and $C$. This view of a delayed communication also clarifies how nodes remain strict regarding their input and output (Definition 3.8), while at the same time cycles are feasible: on delayed edges, input is provided to a node by the queue, allowing the node to start execution independently of the status of the edge's origin node in the current iteration. As an example, consider Figure 3.14, where the same flow graph of Figure 3.11 is depicted, except now with the queue and token based interpretation. The places in the queues (which are shown in finite number in this figure) are illustrated by lines through the edges' arrows and the initial tokens by dots, namely, for one token on edge $e_{\mathrm{TU}}$ and two on $e_{\mathrm{US}}$.

With the introduction of queues and tokens, the state of the computation is at any time reflected by the distribution of the tokens among the queues of the edges. A node is enabled to start execution, to "fire," if each input edge contains a positive number of tokens and each output edge has at least one space in the queue—that is, the node is strict (Definition 3.8). Before starting execution, the node removes a token from every input edge and puts a token on every output edge after finishing.

The token model can be considered a data-driven execution model, as the flow of the tokens invokes the execution of a node without the need for synchronization. Communication between nodes is performed asynchronously with a self-timed execution of the nodes triggered by the flow of the tokens. Moreover, the strict distinction between iterations is also not necessary due to the self-timed execution.

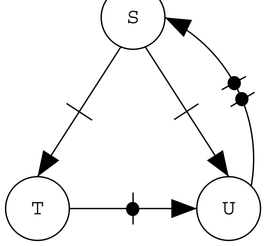


**Figure 3.14.** The flow graph of Figure 3.11 with queues and tokens (initial data items) on the edges; a line through an arrow denotes a place in a queue and a dot denotes an initial data item.

One of the earliest flow graphs for parallel computations is the computation graph (CG) (Karp and Miller [100], Reiter [162]). Each edge has a FIFO queue without restriction of its length and a weight is associated with each edge indicating the initial number of data words in the queue. The token concept is also used in data flow languages (Ackerman [1], Davis and Keller [53]), a graphical approach to software development. Data flow program graphs allow the programming of data flow computers, since both share the self-timed and data-driven nature. The data flow graph (DFG) (Kung [109], Parhi [145]) benefits particularly from the hardware-oriented view of the data-driven execution model, as it is used in VLSI array processor design. Enhanced with two additional weight types, compared to the simple flow graph, the DFG also reflects the computation time of the nodes and the capacity of the FIFO queues. Restricting the queue capacity can lead to deadlocks, which is of real concern in VLSI array processors. DFGs are used for the mapping of parallel computations on VLSI wavefront array processors (Kung [109]).

The data-driven execution model is a valuable mechanism whenever an accurate view of the data flow in space and time is required, as, for example, in hardware-oriented parallelization. For many purposes, however, such a detailed view of the data flow is not necessary and the iteration-driven execution model is appropriate and sufficient. In any case, the flow graph defined in Definition 3.12 is suitable for a data-driven view as soon as the edge delays $D(e)$ are considered initial tokens (Kung [109]), as described earlier. The iteration-driven execution model simply presupposes a queue mechanism for the delayed communication that is sufficiently large, and therefore no further consideration is needed.

### 3.4.2 Summary

The flow graph is an efficient representation of iterative computations, owing to the fact that repetitive parts of computation and communication are modeled only once. On the other side, the flow graph is limited to cyclic computations with usually uniform communication relations (nonuniform communication relations can only be approximated). The granularity of the flow graph is basically inherited from the iterative computation type and is hence fine to medium grained.

Many parallelization techniques for iterative programs are based on the program's representation as a flow graph with or without computation and/or communication costs: unfolding, retiming, software pipelining, mapping, and scheduling, just to name a few. A flow graph with computation and communication costs is simply defined as $FG = (\mathbf{V}, \mathbf{E}, D, w, c)$ (see Definitions 3.9 and 3.12). The flow graphs used by Parhi and Messerschmitt [144, 146] for some of these techniques are called iterative data-flow programs and include computation costs, that is, node weights ($w$). Yang and Fu [208] employ a graph called the iterative task graph (ITG), featuring both computation ($w$) and communications costs ($c$), which is called a communication sensitive data flow graph (CS-DFG) by Tongsima et al. [185]. The signal flow graph (SFG) (Kung [109], Parhi [145]), the counterpart of the DFG (see earlier data-driven execution model), is used for synchronous and uniform computations mapped and scheduled on VLSI systolic array processors, for which consequently neither a data-driven execution model nor the modeling of the computation of communication costs is necessary.

## 3.5 TASK GRAPH (DAG)

This section is dedicated to the task graph, which is used for task scheduling. With the understanding of graph models built in the previous sections, it is straightforward to comprehend and analyze the task graph. During the following discussion, the relationships between the graph models will be developed, including transformation techniques.

In the literature, the task graph is often simply referred to as DAG, which merely describes the graph theoretic properties of the model, namely, that it is a directed acyclic graph. In order to avoid ambiguity with other directed acyclic graphs, the name *task graph* is used in this text.

The discussion starts with the definition of the task graph.

**Definition 3.13 (Task Graph (DAG))** *A task graph is a directed acyclic graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ representing a program $\mathcal{P}$ according to the graph model of Definition 3.7. The nodes in $\mathbf{V}$ represent the tasks of $\mathcal{P}$ and the edges in $\mathbf{E}$ the communications between the tasks. An edge $e_{ij} \in \mathbf{E}$ from node $n_i$ to $n_j, n_i, n_j \in \mathbf{V}$, represents the communication from node $n_i$ to node $n_j$. The positive weight $w(n)$ associated with node $n \in \mathbf{V}$ represents its computation cost and the nonnegative weight $c(e_{ij})$ associated with edge $e_{ij} \in \mathbf{E}$ represents its communication cost.*

During the execution of the program $\mathcal{P}$, every represented task (i.e., node) is executed exactly once. It is presupposed that the program $\mathcal{P}$ is a feasible program, hence the acyclic property of the graph (see Lemma 3.4). No communications or dependences exist in $\mathcal{P}$ other than those reflected by the edges of $G$. The task graph features node and edge weights representing the costs of computation and communication, respectively, with the notable difference that the node weight is here defined as positive rather than nonnegative as in Definition 3.9. This small difference is important for certain properties of the task graph, as will be seen in Section 4.4.

Figure 3.15 illustrates a sample task graph for a small fictitious program; the nodes are named by the letters from $a$ to $k$, while the node and edge weights are noted beside the respective graph elements. This sample graph will be employed as an example throughout the following chapters.

Some of the early scheduling algorithms and those used under restricted conditions (see Chapter 4) employ simplified task graphs, inasmuch as communication and sometimes computation costs are neglected. In that sense, the here defined task graph is a general model, which can be used in these algorithms by either ignoring the computation and/or communication costs or setting them to zero.

The edges of the task graph only reflect flow data dependence (remember, communications establish flow dependence relations). If other dependence relations existed in

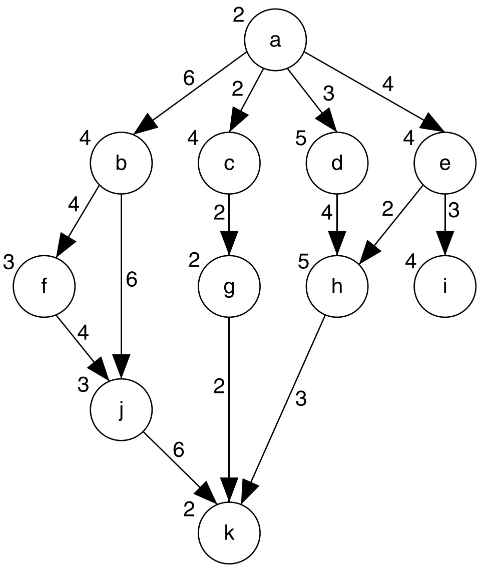


**Figure 3.15.** The task graph for a fictitious program. Nodes are named by letters *a–k*; node and edge weights are noted beside them.

a preliminary version of the program $\mathcal{P}$, they have been eliminated before the construction of the task graph $G$ (the elimination of output and antidependence is explained in Section 2.5.1). For some practical purposes, the difference between the three data dependence types (i.e., flow dependence, antidependence, and output dependence) is not relevant. In such cases, a task graph can be defined, whose edges can represent all types of dependences. The only difference from a DG is then the computation and communication costs of the task graph. However, the definition of "communication costs" for output dependence and antidependence becomes very problematic. Moreover, output dependence and antidependence are provoked by variable reuse. In a distributed system without shared memory, this problem does not exist when the respective tasks are executed on different processors. For these reasons and in order to have one general model, the defined task graph here only represents flow dependences.

The task graph is a general graph model not bound to a particular computation type; that is, it can reflect iterative and noniterative computations. Still, like the DG, it is not adapted to cyclic computations in any way, making the graph size, in terms of nodes, proportional to the number of iterations. That is why the task graph is commonly used for coarse grained, noniterative computations—hence its name task graph. It shares another limitation of the DG with respect to the number of iterations: if the number depends on the input of the program, the task graph cannot be constructed for the general case. In return, the task graph is not limited to uniform interiteration communications as is the (approximation-free) flow graph.

From the definition of the task graph, several similarities with the dependence graph and the flow graph can be observed. The task graph inherits the topology of the

DG for feasible programs. Remember, while the DG is defined as a directed graph (Definition 3.10), Lemma 3.4 demands that it is acyclic in order to represent a feasible program. Consequently, every node is executed only once during the execution of $\mathcal{P}$, which corresponds to the DG model but not to the flow graph. On the other hand, edges reflect communication, as in the flow graph, and not solely dependence relations, as in the DG. Obviously, the reflected communications represent implicitly the real data dependence relations (see Section 2.5.1). A further similarity with the flow graph is the inclusion of computation and communication costs (Definition 3.9), which are contained in several of the flow graph models mentioned (Section 3.4).

The three considered graph models—dependence graph, flow graph, and task graph—share a common basis, which is described by Definition 3.7. As mentioned earlier, there are many additional similarities between the models and it is thus not surprising that it is possible, and common, to convert or transform one model into another (Sinnen and Sousa [173, 174]). The next section will outline some of these techniques, which also helps to gain a general view of graph models.

### 3.5.1 Graph Transformations and Conversions

In the subsequent discussion, a rough distinction will be made between transformations and conversions. Those techniques that require the creation or removal of nodes shall be called transformations and those preserving the original nodes shall be called conversions.

The discussion starts with the conversions, which are sometimes little more than a different interpretation of the given graph model.

***Task Graph to DG*** The conversion of the task graph into a DG is very simple, given that the dependence relations are implicitly expressed through the communication edges. Hence, this conversion is rather an interpretation of the task graph as a DG by ignoring the node and edge weights. It should be noted that the type of data dependence reflected by the edges is *flow*, or real, dependence. Other dependence relations do not exist by definition of the task graph. For this reason, the opposite conversion, from DG to task graph, is in general not valid, because dependence graphs can comprise all types of dependence.

***Extracting Iterative Kernel—Flow Graph to Task Graph*** Even though the task graph is not designed specifically for the representation of cyclic computations like the flow graph, it proves useful for the representation of the kernel of the iterative computation—the loop body. If one iteration consists of more than one task, surely a task graph can be constructed to represent the iteration's tasks and their communications. The same task graph can also be obtained by converting the flow graph that represents the entire iterative computation. The simple conversion is performed by removing all edges from the graph that have nonzero delays, that is, $\{e \in \mathbf{E} : D(e) \neq 0\}$: in other words, all edges that represent interiteration communications. Obtained are the nodes connected only by the intraiteration communication

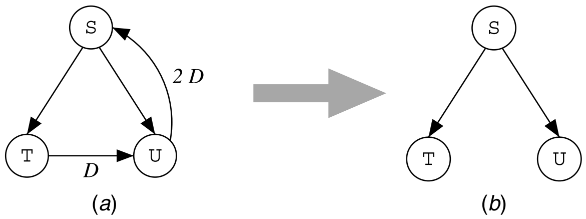


**Figure 3.16.** (*a*) The flow graph of Figure 3.11 is converted into the task graph of the loop kernel (*b*) by removing the delayed edges.

edges, which is the task graph of the iterative kernel. As an example, this conversion is performed with the sample flow graph of Figure 3.11 of the previous section, shown here in Figure 3.16(*a*). By removing the nonzero communication edges, the graph illustrated in Figure 3.16(*b*) is obtained. The correctness of the task graph is verified by analyzing the code of the represented program of Example 10: the output of `S` ($A(i)$) is the input of `T` and `U` creating the two intraiteration communications.

Using a task graph for modeling the iterative kernel is successfully employed for the scheduling of cyclic computations (Sandnes and Megson [164], Sandnes and Sinnen [166], Yang and Fu [208]). The major advantage is that once the iterative computation is represented by a task graph, task scheduling, as discussed in the following chapters, can be utilized. Note that the number of iterations does not need to be known for this technique.

***Unrolling—Flow Graph to Task Graph*** An alternative technique to create a task graph from a flow graph is unrolling or unfolding, which was already used for the proof of Lemma 3.5. Unrolling, according to the above loose definition, is a transformation since it creates a new graph. In contrast to the previous conversion, not only is the loop kernel represented by the task graph but the entire iterative computation is. It follows that the number of iterations must be known at the time of construction—a condition that is not always fulfilled at compile time, since the number of iterations can depend on the input of the computation. Recall that the flow graph is constructed independently of the number of iterations (Section 3.4).

To construct the unrolled task graph, a graph is created consisting of the respective nodes and edges of the loop-body task graph for every iteration. A node is identified by its name in the flow graph and the iteration index to which it belongs. Those nodes and edges already reflect all task instances of the iterative computation and their intraiteration communications. The remaining interiteration communication edges are added to the graph by considering each node of the new graph and adding edges for its leaving interiteration communications, as long as the iteration index of the destination node is part of the computation. If the considered node $n$ is of iteration $i$, the edge $e$ is created between $n$ and the respective destination node of iteration $i + D(e)$, unless $i + D(e) > N$ for $1 \leq i \leq N$, with $N$ being the number of iterations. The node and edge weights representing the computation and communication costs, respectively, are set to the values of the corresponding flow graph elements.

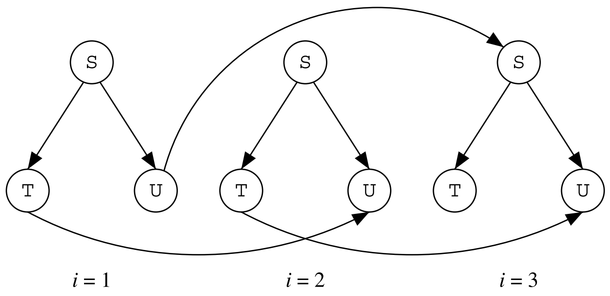


**Figure 3.17.** The unrolled flow graph of Figure 3.16($a$) for three iterations.

Figure 3.17 shows the unrolling example of Section 3.4, where the flow graph of Figure 3.16($a$) is unrolled for three iterations (i.e., $N = 3$) in the underlying code of Example 10. One clearly sees the three distinct kernel task graphs for the three iterations and the three interiteration communications. There is no leaving edge from node U in the second iteration, since the potential destination node (in iteration 4) is not part of the computation. The same holds for the interiteration communications of the nodes of the third iteration.

A graph constructed from a flow graph without the attribution of weights to the nodes and edges can be interpreted as the dependence graph—reflecting only flow data dependences—of the iterative computation. This close relationship between task graph and DG was examined earlier. More precisely, the DG obtained by unrolling a flow graph is the iteration DG of the computation, comprising only uniform dependence relations.

Sometimes the unrolling is done only for a fraction of the total number of iterations. This partial unrolling, which is sufficient for some purposes (Sandnes and Megson [164], Yang and Fu [208]), has two advantages: (1) the total number of iterations does not need to be known and (2) the size of the unrolled graph, in terms of the number of nodes, is not proportional to the number of iterations. However, the resulting graph remains a flow graph; it is not a task graph. For this reason, partial unrolling is at times employed as a prestage to the extraction of the iterative kernel as described earlier (e.g., Yang and Fu [208]).

***Projection—Iteration DG to Flow Graph*** The countertechnique to unrolling is the projection of an iteration DG to a flow graph (Kung [109]). Multiple equal nodes of the iteration DG (i.e., nodes that represent the same type of task) are projected into one node of the flow graph. An illustrative example is the projection of a two-dimensional iteration DG into a flow graph, reducing the iteration DG by one dimension. Figure 3.18 shows such a projection for the two-dimensional iteration DG of Figure 3.10 along the $i$-axis, resulting in the depicted flow graph.

Essentially, the projection is performed by merging all nodes along the projection direction into one node and by transforming the communication edges into new edges with delays. A delay substitutes the spatial component of the distance vector of the

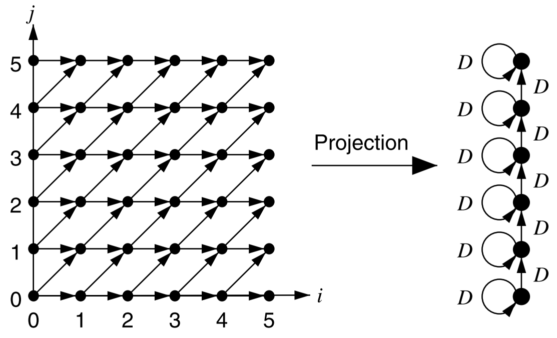


**Figure 3.18.** The iteration DG of Figure 3.10 is projected along the *i*-axis into a flow graph.

edge that is parallel to the direction of the projection. In other words, a spatial dimension of the distance vectors is transformed into a temporal one. In the example, the *i*-dimension is transformed into the temporal dimension of the delays.

The projection in the above example is linear along one of the axes of the iteration DG. In general, the projection is not required to be along an axis, in fact, it is not even required to be linear (Kung [109]). However, an inherent iterative structure must be present in the DG; otherwise the computation cannot be described as a flow graph, so normally a general DG cannot be transformed into a flow graph.

A common application of projection is in VLSI array processor design (Kung [109]), where the iteration DG often serves as an initial model to obtain the flow graph, which is a description of the application closer to the hardware level.

The conversions and transformations demonstrate the close relationships of the various graph models. To conclude and summarize the discussion of these techniques, the relationships are illustrated in Figure 3.19. Shown are the three major graph models—DG, flow graph, and task graph—linked by the conversions and transformations discussed in this section.

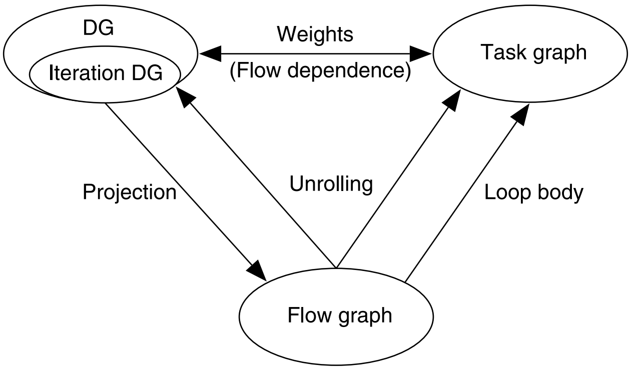


**Figure 3.19.** The three major graph models and the relationships among them.

### 3.5.2 Motivations and Limitations

This subsection discusses the motivations for the adoption of the task graph model for task scheduling, which is accompanied by a critical discussion of its limitations. Basically, this goal is achieved by summarizing and comparing the principal properties of the presented graph models.

***Motivations*** So far, this chapter has demonstrated that the abstraction of a program as a graph can very well capture the dependence and communication structure of a program. The task graph model has several properties that make it particularly suitable for task scheduling.

- *General.* It is desirable that the graph model is as general as possible, in terms of the computation types, and the task graph is such a model. In comparison, the flow graph is restricted to iterative computations with uniform communications (otherwise it is not accurate).
- *Simple.* The task graph's focus on communications, which correspond to real data dependence, permits task scheduling to concentrate on these essential precedence constraints. Other dependences reflected in the DG can be eliminated and are not inherent to the represented computation.
- *Modeling of Computation and Communication Costs.* Scheduling algorithms for modern parallel systems must be aware of the computation and communication costs. It is therefore crucial that they are represented in the graph model; hence, a DG does not suffice.
- *Close Relationship to Other Models.* The previous section outlined the various relationships between the discussed models. In connection with a conversion or transformation, techniques and algorithms based on the task graph can be employed for other models.

***Limitations*** The task graph as a general model does not provide any mechanism to efficiently represent an iterative computation.

- *Iterative Computations.* For iterative computations, the size of the task graph depends on the number of iterations, which directly influences the memory consumption and the processing time of task scheduling algorithms. With the loss of regularity information in the task graph, scheduling algorithms also cannot benefit from the inherent regularity of cyclic computations. Furthermore, if the number of iterations is only known at runtime, the task graph cannot be constructed for the general case. Still, scheduling techniques for cyclic computations (Sandnes and Megson [164], Sandnes and Sinnen [166], Yang and Fu [208]) do use the task graph and associated techniques, for example, to represent the iterative kernel.

Another limitation is not a particular limitation of the task graph, but of all models covered in this chapter. In fact, it was already introduced during the definition of the general graph model in Section 3.2.

- *Static Model.* The graph models according to Definition 3.7, to which the task graph model belongs, do not exhibit conditional statements of the code; that is, there is no branching. These control dependences are either transformed into data dependences or encapsulated within a node (see Section 3.2).

### 3.5.3 Summary

The task graph is the graph model of choice for task scheduling. It clearly exhibits the task and communication structure of a program, while also reflecting the computation and communication costs. Its properties were summarized in the previous section, when analyzing the motivations for the model's choice and its limitations. The general nature of the task graph, together with the typically coarse granularity of the represented tasks, indicates the adequacy of employing the task graph for distributed parallel systems with SPMD or MIMD streams.

Chapter 4 returns to the task graph model after establishing a basic understanding of the task scheduling problem on parallel systems. It is there that the task graph model and its properties are examined further in the context of task scheduling.

Also, Chapter 4 addresses the computation and communication costs associated with the nodes and the edges of the task graph, respectively. Until now, the node and edge weights were introduced only as abstract notations of computation and communication costs. Evidently, such costs are related to the target parallel system on which the program represented by the task graph is executed. Thus, it is necessary to define the target parallel system model, before the concept of costs in the task graph can be substantiated.

## 3.6 CONCLUDING REMARKS

This chapter presented and analyzed in depth the three major graph models for the representation of computer programs: dependence graph, flow graph, and task graph. It started with the fundamental concepts of graph theory and then formulated a common foundation for graph models representing computer programs. With this background, the three models and their properties were discussed.

While this chapter serves as an introduction to graph models for program representation, its objective was to introduce and analyze the task graph model of task scheduling. This broad approach was deemed crucial in order to establish a comprehensive understanding of the task graph. It was shown that the task graph model inherits the structure of the dependence graph and many of its properties. Furthermore, the flow graph, as a concise representation of iterative computation, was related to the task graph by means of transformations and conversions, discussed in

Section 3.5.1. At the end, the motivations for the task graph model were presented and its limitations were analyzed.

The chapter's focus on the task graph resulted in the omission of several other, less common, graph models, some of which are analyzed by Sinnen and Sousa [173]:

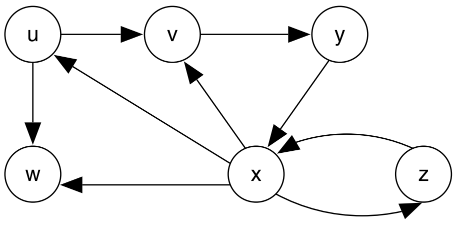
- *Task Interaction Graph* (*TIG*) (Stone [182]). An undirected graph model only capturing the communication relations between entire processes as opposed to tasks. The TIG is used for mapping of processes onto parallel processors (e.g., Sadayappan et al. [163], Stone [182]).
- *Temporal Communication Graph* (*TCG*) (Lo [129]). A model that integrates the task and process oriented view of a parallel program. The TCG is based on the space–time diagram introduced by Lamport [116] and can be interpreted for mapping and scheduling algorithms as a TIG as well as a task graph (Lo et al. [130]).
- *Control Flow Graph* (*CFG*) *and Control Dependence Graph* (*CDG*) (Allen and Kennedy [12], Wolfe [204]). These two closely related directed graph models represent the control flow and control dependence of a program. As a consequence, they reflect conditional execution paths (branching, see Section 3.2), as opposed to all other graph models discussed in this chapter. Both are used in compilers to analyze and handle the control flow of a program.

For the fundamental problem of integrating iterative and noniterative computations into one efficient graph model, the idea of a hierarchical graph (Sinnen and Sousa [175]) emerged, where a node of a higher level can represent an entire subgraph.

With the end of this chapter, the foundation for the discussion of task scheduling in the next chapter has been laid.

## 3.7 EXERCISES

**3.1** Section 3.1 reviews basic graph concepts for undirected and directed graphs. Consider the following directed graph $G = (\mathbf{V}, \mathbf{E})$ consisting of 6 vertices and 9 edges:



**(a)** Which vertex has the highest degree?

**(b)** What is the average in-degree of the vertices? What is the average out-degree?

**(c)** Is vertex $y$ an ancestor or a descendant of vertex $x$?

**(d)** Is vertex $v$ reachable from vertex $w$? Is vertex $w$ reachable from vertex $v$?

**3.2** Paths and cycles are important and powerful concepts in graphs (Section 3.1). Consider again the directed graph of Exercise 3.1.

**(a)** How many simple cycles has the graph? Specify them.

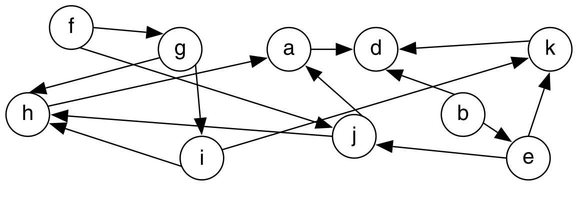
**(b)** What is the length of the longest simple path, in terms of the number of edges, in this graph? Specify a path with such a length.

**3.3** Commonly, there are two different approaches to the representation of a graph in a computer (Section 3.1.1). Consider again the directed graph of Exercise 3.1.

**(a)** Give an adjacency matrix representation of this graph.

**(b)** Give an adjacency list representation of this graph.

**3.4** The topological order is an important concept for directed acyclic graphs (Section 3.1.2). Find a topological order for the following directed acyclic graph $G = (\mathbf{V}, \mathbf{E})$ consisting of 10 vertices:



Which vertices are source (entry) vertices and which are sink (exit) vertices?

**3.5** Construct the dependence graph for the code fragment of Exercise 2.9.

**3.6** Construct the iteration dependence graph for the double loop of Exercise 2.11.

**3.7** Construct the flow graph (Section 3.4) for the following iterative computation:

```
for i = 1 to n do
  S: A(i) = B(i) - 1
  T: B(i+1) = (A(i-1)-A(i))/2
  U: C(2 * i) = B(i-1) + E(i+2)
  V: D(i) = C(2 * i-2) + B(i+1)
end for
```

**3.8** Construct a task graph (Section 3.5) for the code below. Each line shall be represented by one task, named by its line number, and the costs shall be assumed as follows:

- *Computation.* Assignment alone: 1 unit; add/subtract operation: 2 units; multiply operation: 3 units; divide operation: 4 units.
- *Communication.* Communicating a variable with a small letter and with a capital letter costs 1 unit and 2 units, respectively (imagine variables with capital letters to have higher precision).

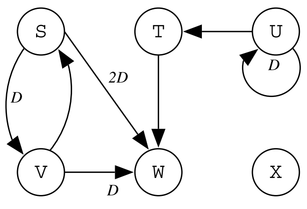
```
a = 56
b = a * 10 + 2
C = (b - 2) / 3
D = 91.125
E = D * a
F = D * b + 1
g = 11 + a
H = (E + F) * g
```

**3.9** Construct a task graph (Section 3.5) for the code below. Take alternatively a coarse grained or a fine grained approach and make clear which part of the code is represented by which node of the task graph. It might be useful to convert control dependence into data dependence (Section 3.2). The costs shall be assumed as follows:

- *Computation.* Assignment alone: 1 unit; add/subtract operation: 2 units; multiply operation: 3 units; function call: 25 units; variable comparison: 3 units (conditional execution: 1 unit).
- *Communication.* Communicating one variable or one array element costs 1 unit.

```
{Input: arrays A and B of size 10}
A(1) = A(10) + 2 * B(1)
for i = 2 to 10 do
  A(i) = A(i-1) + 2 * B(i)
end for
x = function(A)
y = function(B)
if x > y then
  result = x - y
else
  result = y - x
end if
```

**3.10** Given is the following flow graph $FG = (\mathbf{V}, \mathbf{E}, D)$ consisting of 6 nodes:



Extract the task graph representing the iterative kernel (Section 3.5.1). To simplify, assume that all computation and communication costs are of unit size and can therefore be neglected.

**3.11** Unrolling is a technique used to transform generally cyclic flow graphs into acyclic task graphs (Section 3.5.1). Consider again the flow graph of Exercise 3.10. Unroll this flow graph into a task graph for 4 iterations. To simplify, assume that all computation and communication costs are of unit size and can therefore be neglected. Name the nodes of the task graph using their original flow graph name and a suffix indicating the iteration number (e.g., `S`.2 for node `S` of iteration 2).

**3.12** As discussed in Section 3.5.1, unrolling has two disadvantages: (1) the number of iterations must be known and (2) the size of the resulting task graph grows with the number of iterations. Partial unrolling is a technique that avoids both problems by only unrolling the graph for a small, fixed number of iterations, which is lower than the total number of iterations. Consider again the flow graph of Exercise 3.10.

**(a)** Partially unroll this flow graph for 2 iterations; that is, the total number of iterations $N$ is higher than 2. It can be assumed that the number of total iterations $N$ is always an even number. Remember, the resulting graph remains a flow graph.

**(b)** Extract the task graph representing the iterative kernel (Section 3.5.1) of the unrolled flow graph (see also Exercise 3.10).

**3.13** The purpose of Exercises 3.5 – 3.9 was to construct a graph from a given code fragment. In this exercise it is the other way around: the task is to write a simple code fragment that corresponds to a given graph. The code should only consist of array variables, assignments, and add operations, while the number of iterations is given as $N$. Initialization of variables can be ignored. Write a simple code fragment whose flow graph is identical to:

**(a)** The flow graph of Exercise 3.10.

**(b)** The partially unrolled flow graph of Exercise 3.10.

**3.14** In Exercise 3.11 the purpose was to unroll a flow graph into a task graph. This exercise is about reconstructing the original flow graph from an unrolled task graph (costs are neglected). Let the acyclic directed graph below be an unrolled flow graph for 4 iterations.

$$\begin{tikzcd}
a \arrow[r] & c \arrow[r] \arrow[d] \arrow[rrrd] & e \arrow[r] & f \arrow[r] \arrow[d] \arrow[rrrd] & i \arrow[r] & k \arrow[r] \arrow[d] & m \arrow[r] & o \arrow[d] \\
b \arrow[r] \arrow[rr, bend right] & d & g \arrow[r] \arrow[rr, bend right] & h & j \arrow[r] \arrow[rr, bend right] & l & n \arrow[r] & p
\end{tikzcd}$$

Reconstruct the original flow graph.