Chapter 1

Dynamic Programming

1.1 Definitions of DP formulation

We have investigated several discrete optimization techniques, algorithm in the previous chapters. The *Dynamic programming* (DP) defines a consequent way how the problems can be treated and classifies them. Nowadays, DP is a commonly used technique for solving discrete optimization problems such as scheduling, string-editing, packaging, and inventory management. As we will see, DP views a problem as a set of interdependent subproblems. It solves subproblems and uses the results to solve larger subproblems until the entire problem is solved. The solution to a subproblem is expressed as a function of solutions to one more subproblems at the preceding levels.

Before discussing the basic formal definitions of DP let we see an example. We give the DP formulation of the problem of finding a shortest path between a pair of vertices in an acyclic graph. (Refer to Section of Graph Algorithms.) (See Fig.??.)

An edge connecting node i to node j has cost $c(i, j)$. If two vertices i and j are not connected then $c(i, j) = \inf$. The graph contains n nodes numbered $0, 1, \ldots, n-1$, and has an edge from node i to node j only if $i < j$. The shortest-path problem is to find a least-cost path between nodes 0 and $n-1$. Let $f(x)$ denote the cost of the least-cost

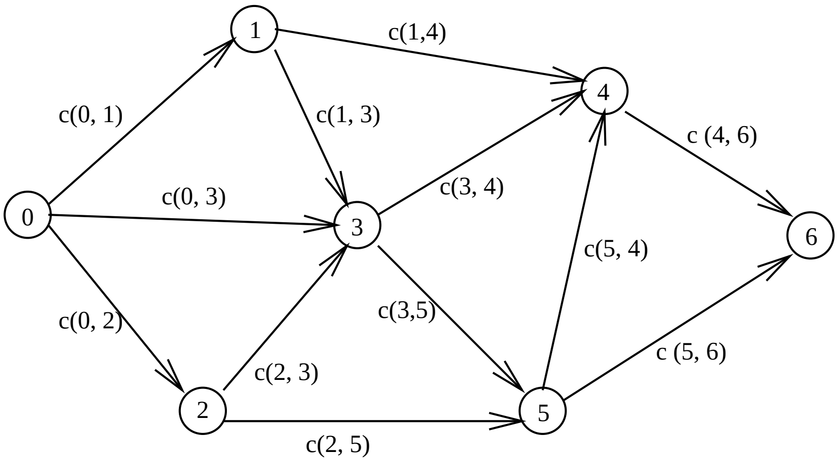

Figure 1.1: Example graph

path from node 0 to node x. Thus, $f(0)$ is zero, and finding $f(n-1)$ solves the problem. The DP formulation for this problem yields the following recursive equations for $f(x)$:

$$f(x) = \begin{cases} 0 & x = 0 \\ \min_{0 \leq j < x}\{f(j) + c(j,x)\} & 1 \leq x \leq n-1 \end{cases}$$

In general, the solution to a DP problem is expressed as a minimum (or maximum) of possible alternative solutions. Each of these alternative solutions is constructed by composing one or more subproblems.

In the example of Fig.?? we can compute the $f(x_0)$ solution by the following sequence:

$$\begin{aligned}
f(x_6) &= 0 \\
f(x_5) &= \min\{f(x_4) + c(5,4), c(5,6)\} \\
f(x_4) &= c(4,6) \\
f(x_3) &= \min\{f(x_4) + c(3,4), f(x_5) + c(3,5)\} \\
f(x_2) &= \min\{f(x_3) + c(2,3), f(x_5) + c(2,5)\} \\
f(x_1) &= \min\{f(x_3) + c(1,3), f(x_4) + c(1,4)\} \\
f(x_0) &= \min\{f(x_1) + c(0,1), f(x_2) + c(0,2), f(x_3) + c(0,3)\}
\end{aligned}$$

If r represents the cost of a solution composed of subproblems $x_1, x_2, \ldots, x_i$ then r can be written as

$$r = g(f(x_1), f(x_2), \ldots, f(x_l)).$$

The function g is called the *composition function*, and its nature depends on the problem. If the optimal solution to each problem is determined by composing optimal solutions to the subproblems and selecting the minimum (or maximum), the formulation is said to be a DP formulation.

DP represents the solution to an optimization problem as a recursive equation whose left side is an unknown quantity and whose right side is a minimization (or maximization) expression. Such an equation is called a *functional equation* or an *optimization equation.*

In the previous equation the composition function g is given by $f(j) + c(j, x)$. This function is additive, since it is sum of two terms. In a general DP formulation, the cost function need not be additive.

A functional equation that contains a single recursive term (for example, $f(j)$ yields a *monadic* DP formulation. For an arbitrary DP formulation, the cost function may multiple recursive terms. DP formulations whose cost function contains multiple recursive terms are called *polyadic* formulations.

The dependencies between subproblems in a DP formulation can be represented by a directed graph. Each node in the graph represents a subproblem. A directed edge from node i to node j indicates that the solution to the subproblem represented by node i is used to compute the solution to the subproblem represented by node j. If the graph is acyclic, then the nodes of the graph can be organized into levels such that subproblems at a particular level depend only on subproblems at previous levels. In this case, the DP formulation can be categorized as follows. If subproblems at all levels depend only on the results at the immediately preceding levels, the formulation is called *serial*, in the other, more complicated cases we call it *nonserial* DP formulation.

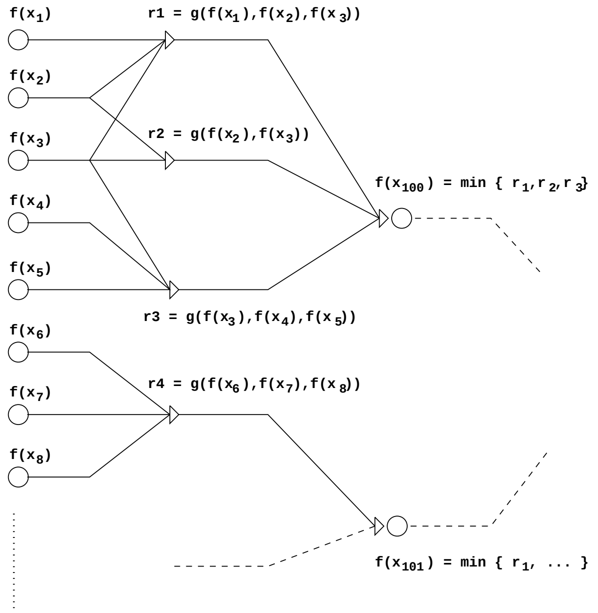

Figure 1.2: Serial, polyadic DP formulation

Based on the preceding classification criteria, we define four classes of DP formulations: *serial monadic, serial polyadic, nonserial monadic*, and *nonserial polyadic*. These classes, however, are not exhaustive; some DP formulations cannot be classified into any of these categories.

In Fig.?? we illustrated how DP defines the problems. The solution to problem x_{100} is the minimum of the three possible solutions having costs r_1, r_2, and r_3. The cost of the first solution is determined by composing solutions to subproblems x_1 and x_2, the second solution by composing solutions to subproblems x_2 and x_3, and the third solution by composing solutions to subproblems x_3, x_4 and x_5. However the solutions x_{100} and x_{101} can serve as subproblems solution to another bigger problem, and cetera, till we found the solution of the main, original problem.

This example is serial DP formulation while 'levels' can be defined and it recursively uses only the subproblems of the preceding levels. On the other side it is polyadic, while in the composition function more than one subproblems appear.

We can cover a wide variety of problems by DP approach, that is why it is difficult to find generic parallel algorithms for them. In the forthcoming sections we systematically go through the different classes of DP and discuss parallel DP formulations for sample problems in each class.

1.2 Serial Monadic DP Formulations

Two representative problems of serial monadic DP formulation is discussed in this chapter section discusses the shortest-path problem for a multistage graph and the 0/1 knapsack problem. The former is already treated in the Graph Algorithms chapter.

1.2.1 The Shortest-Path Problem

A multistage graph of $r+1$ levels is a graph where the nodes are arranged in levels, as shown in Fig.??. Each node at level i is connected to every node at level $i+1$. Levels zero and r contain only one node, and every other level contains n nodes. We refer to the node at level zero as the starting node S and the node at level r as the terminating node R.

The i^{th} node at level l in the graph is labeled v_i^l. The cost of an edge connecting v_i^l to node v_j^{l+1} is labeled $c_{i,j}^l$.

The (minimal) cost of reaching the goal node R from any node v_i^l is represented by C_i^l. If there are n nodes at level l, the vector $[C_1^l, \ldots, C_{n-1}^l]^T$ is referred to as $\mathbf{C}^l$.

The objective of this problem is to find the shortest path from S to R. We can use one of the general algorithm explained in the Graph Algorithms chapter, but we give here another formulation with DP which extensively exploits the special structure (i.e. levels) of the graph.

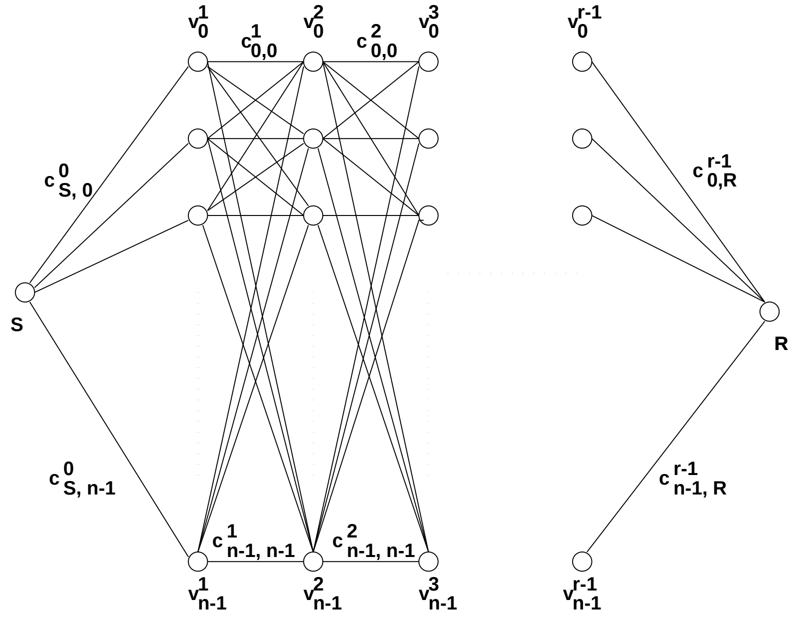

Figure 1.3:

Hence, the shortest-path problem reduces to computing $\mathbf{C}^0$. Since the graph has only one starting node, $\mathbf{C}^0 = [C_0^0]$. The structure of the graph is such that any path from v_i^l to R includes a node $V_j^{l+1}(0 \leq j \leq n-1)$.

Quite similarly to the example what we have seen in the first section, the cost of any such path is the sum of the cost of the path between v_i^l and v_j^{l+1} and the cost of the shortest path between v_j^{l+1} and R (which is given by C_j^{l+1}).

Thus, C_i^l, the cost of the shortest path between v_i^l and R, is equal to the minimum cost over all paths through each node in level $l+1$. Therefore,

$$C_i^l = \min\{(c_{i,j}^l + C_j^{l+1}) \mid j \text{ is a node at level } l+1\}.$$

In the special case when $l = r-1$, all nodes v_j^{r-1} have only one edge connecting them to the goal node R at level r, the cost C_j^{r-1} is equal to $c^{r-1}_{j,R}$, which implies

$$\mathbf{C}^{r-1} = [c^{r-1}_{0,R}, c^{r-1}_{1,R}, \ldots, c^{r-1}_{n-1,R}].$$

The recursive equation contains only one recursive term in its right-hand side, it is a monadic formulation. Note that the solution to a subproblem requires solutions to subproblems only at the immediately preceding level. Consequently, this is a serial monadic formulation.

Using this recursive formulation of the shortest-path problem, the cost of reaching the goal node R from any node at level $l (0 < l < r-1)$ is

$$\begin{aligned}
C^l_0 &= \min\{(c^l_{1,0} + C^{l+1}_0), (c^l_{0,1} + C^{l+1}_1), (c^l_{1,n-1} + C^{l+1}_{n-1})\}, \\
C^l_1 &= \min\{(c^l_{1,0} + C^{l+1}_0), (c^l_{1,1} + C^{l+1}_1), (c^l_{1,n-1} + C^{l+1}_{m-1})\}, \\
\vdots &= \vdots \\
C^l_{n-1} &= \min\{(c^l_{n-1,0} + C^{l+1}_0), (c^l_{n-1,1} + C^{l+1}_1), (c^l_{n-1,n-1} + C^{l+!}_{n-1})\}.
\end{aligned}$$

We can consider the operation of multiplying a matrix with a vector. For that purpose we have to replace the addition operation by minimization and the multiplication operation by addition. The former set of equations can be written as

$$\mathbf{C}^l = M_{l,l+1} \times \mathbf{C}^{l+1},$$

where $\mathbf{C}^l$ and $\mathbf{C}^{l+1}$ are $n \times 1$ vectors representing the cost of reaching the goal node from each node at levels l and $l+1$, and $M_{l,l+1}$ is an $n \times n$ matrix in which entry (i,j) stores the cost of the edge connecting node i at level l to node j at level $l+1$. This matrix is

$$M_{l,l+1} = \begin{bmatrix}
c^l_{0.0} & c^l_{0,1} & \cdots & c^l_{0,n-1} \\
c^l_{1.0} & c^l_{1,1} \cdots & c^l_{1,n-1} & \\
\vdots & \vdots & \vdots & \\
c^l_{n-1,0} & c^l_{n-1,1} & \cdots & c^l_{n-1,n-1}
\end{bmatrix}$$

The shortest path problem has thus been reformulated as a sequence of matrix- vector multiplications. In a sequential algorithm we compute first $\mathbf{C}^{r-1}$ while it is obvious as we have seen it, and then recursively compute the $\mathbf{C}^{r-2}, \mathbf{C}^{r-3}, \ldots, \mathbf{C}^{1}$ by using the previous matrix equation. Finally, $\mathbf{C}^{0}$ is computed using our defining recursive equation.

Since there are n nodes at each level, the cost of computing each vector $\mathbf{C}^{l}$ is $O(n^2)$. The parallel algorithm for this problem can be derived using the parallel algorithms for the matrix-vector product discussed in the Numerical Algorithms chapter.

For example, on a mesh, $O(n)$ processors can compute each vector $\mathbf{C}^{l}$ in $O(n)$ time and solve the entire problem in $O(rn)$ time.

This example can serve as a general template for many serial monadic DP problems. The serial and monadic feature assures us to have a dependency graph structure which yields to this matrix formulation. However there are cases when this approach is not effective while, for example the formulation produces a sparse matrix. In such cases other parallel algorithmic methods (e.g. for sparse matrices) have to be applied.

1.2.2 The 0/1 Knapsack Problem

The well known one-dimensional 0/1 knapsack problem is a problem of a tourist who wants to pack the most useful objects in his bag, but of course the capacity of the bag is limited. To be more formal, let be the capacity of the knapsack c and a set of n objects numbered $1, 2, \ldots, n$. Each object i has weight w_i and profit p_i. Object profits and weights are integers. Let $v = [v_1, v_2, \ldots, v_n]$ be a solution vector in which $v_i = 0$ if object i is not in the knapsack, and $v_i = 1$ if it is in the knapsack. The goal is to find a subset of objects to put into the knapsack so that

$$\sum_{i=1}^{n} w_i v_i \leq c$$

(that is, the object fit into the knapsack) and

$$\sum_{i=1}^{n} p_i v_i$$

is maximized (that is, the profit is maximized).

The first idea could be to solve this problem is to consider all 2^n possible subsets of the n objects and choose the one that fits into the knapsack and maximizes the profit. It leads, of course, in practical cases to a non computable problem due to the huge number of combinatorial cases.

Here we provide a DP formulation that is faster than the simple method when $c = O(2^n/n)$. Let $F[i,x]$ be the maximum profit for a knapsack of capacity x using only objects $\{1,2,\dots,i\}$. Then $F[n,c]$ is the solution to the problem. The DP formulation for this problem is as follows:

$$F[i,x] = \begin{cases} 0 & x \geq 0, i = 0 \\ -\inf & x < 0, i = 0 \\ \max_{1 \leq i \leq n}\{F[i-1,x], (F[i-1,x-w_i] + P-i)\} & 1 \leq i \leq n \end{cases}$$

This recursive equations yields a knapsack of maximum profit. When the current capacity of the knapsack is x, the decision to include object i can lead to one of two situations:

- the object is not included, knapsack capacity remains x, and profit is unchanged;
- the object is included, knapsack capacity becomes $x - w_i$, and profit increases by p_i.

The DP algorithm decides whether or not to include an object based on which choice leads to maximum profit.

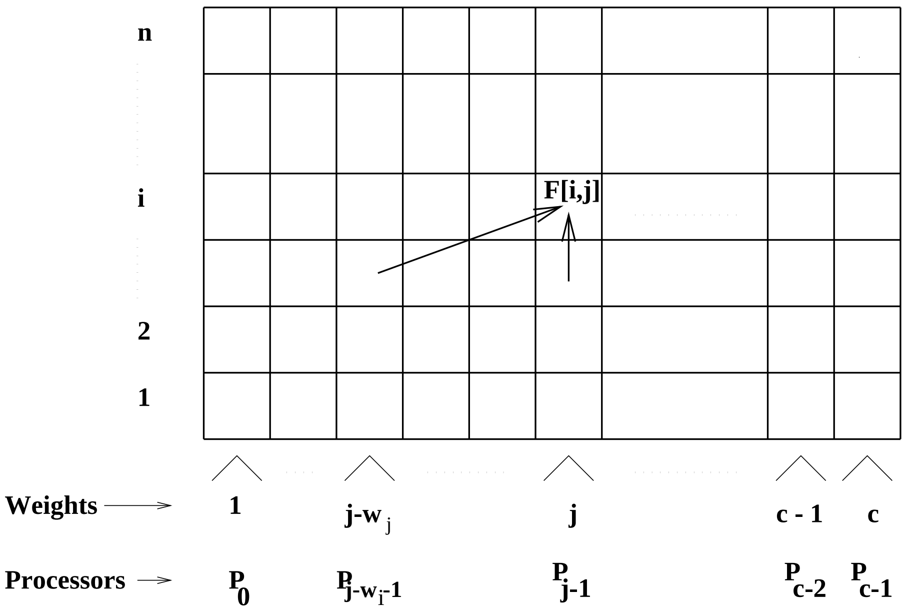

Figure 1.4:

The sequential algorithm for this DP formulation maintains a table F of size $n \times c$. (See Fig.??.) The table contains all $F[i,j]$ value. We exploit here that the possible weights are integers ranging from 0 1 to the capacity c. The first (bottom) row belong to the case when only the first object is selected. The algorithm first determines the maximum profit by using the first object with knapsacks of different capacities. Filling entries in subsequent rows requires two entries from the previous row: one from the same column and one from the column offset by the weight of the object. Thus, the computation of an arbitrary entry $F[i,j]$ requires $F[i-1,j]$ and $F[i-1,j-w_i]$. This is depicted in Fig.??.

Obviously, Computing each entry takes constant time (less than c steps), so the sequential run time of this algorithm is $O(nc)$.

It is trivial, that this formulation is a serial, while the subproblems $F[i,x]$ are organized into n levels for $i = 1, 2, \ldots, n$. Computation of problems in level i depends only on the subproblems at level $i-1$. The formulation is monadic because each of the two alternate solutions of $F[i,x]$ depend on only one subproblem, namely the $F[i-1,x]$ function, where x is the variable of the function.

Let us use CREW PRAM model for the parallel formulation of this algorithm with processors labeled P_0 to P_{c-1}.

Processor P_{r-1} computes the r^{th} column of matrix F. When computing $F[j,r]$ during iteration j, processor P_{r-1} require the values $F[j-1,r]$ and $F[j-1,r-w_i]$. Processor P_{r-1} can read any element of matrix F in constant time, so computing $F[j,r]$ also requires constant time. Therefore, each iteration takes constant time. Since there are n iteration, the parallel run time is $O(n)$. The formulation uses c processors, hence, its processor-time product is $O(nc)$. Therefore, the algorithm is cost-optimal.

If we use a hypercube architecture the formulation is (generally) not cost optimal. Let us consider c-processor hypercube. Table F is distributed among the processors so that each processor is responsible for one column. We have depicted the the mapping on the Fig.**??**.

Each processor locally stores the weights and profits of all objects. In the j^{th} iteration, for computing $F[j,r]$ at processor P_{r-1}, $F[j-1,r]$ is available locally but $F[j-1,r-w_j]$ must be fetched from another processor. This circular w_j-shift operation described in the Fundamentals chapter. The time taken by a circular shift operation on a hypercube with cut-through routing has an upper bound of $(t_s + t_w + t_h \log c)$ (since only one word is being shifted). If the sum and maximization operations take time t_c, then each iteration takes time $t_c + t_s + t_w + t_h \log c$. Because there are n such iterations, the total time is given by $O(n \log c)$. The processor-time product for this formulation is $O(nc \log c)$; therefore, the algorithm is not cost-optimal.

Let us see what happens to this formulation as we increase the number of elements per processor. Using a p-processor hypercube, each processor computes c/p elements of the table. In the j^{th} iteration, processor P_0 computes the values of elements $F[j,1], \ldots, F[j,c/p]$, processor P_1 computes values of elements $F[j,c/p+1, \ldots, F[j,2c/p]$, and so on. Computing the value of $F[j,k]$, for any k, requires values $F[j-1,k]$

and $F[-1, k - w_j]$. Required values of the F table can be fetched from remote processors by performing a circular shift. Depending on the values of w_j and p, the required nonlocal values may be available from one or two processors. The time for this operation is at most $(2t_s + t_w c/p + 2t_h \log p)$. Since each processor computes c/p such elements, the total time taken by a processor for each iteration is $t_c c/p + 2t_s + t_w c/p + 2t_h \log p$. Therefore, the parallel run time of the algorithm for n iterations is $n(t_c c/p + 2t_s + t_w c/p \log p)$. In asymptotic terms, this algorithm's parallel run time is $O(nc/p + n \log p)$. Its processor-time product is $O(nc + np \log p)$, which cost-optimal for $c = \Omega(p \log p)$.

There is an upper bound on the efficiency of this formulation. The reason is that the amount of data that needs to be communicated is of the same order as the amount of computation at each processor. This upper bound is determined by the values of t_w and t_c. Mapping this parallel algorithm onto a mesh results in a non-cost-optimal formulation.

1.3 Nonserial Monadic DP Formulations

The DP algorithm for determining the longest common subsequence of two given sequences can be formulated as a nonserial monadic DP formulation.

1.3.1 The Longest-Common-Subsequence Problem

Given two sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, and $b = \langle a'_1, a'_2, \ldots, a'_k \rangle$. We call B a subsequence of A if we can produce B by deleting entries from A. The Fig**??** shows an example: $\langle a, b, z \rangle$ is a subsequence of $\langle c, a, d, b, r, z \rangle$, but $\langle a, c, z \rangle$ and $\langle a, d, l \rangle$ are not. The *longest-common-subsequence* (LCS) problem can be stated as follows. Given two sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$, find the longest

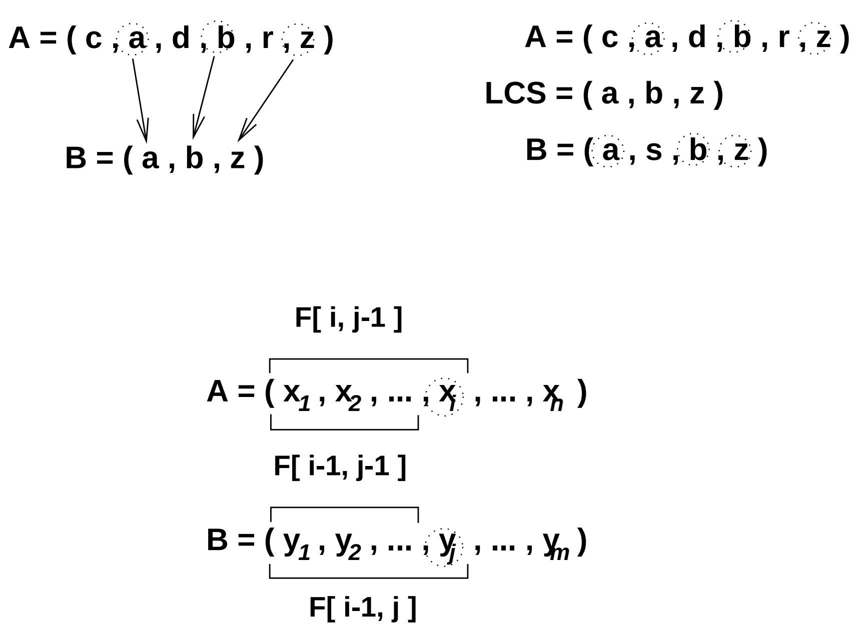

Figure 1.5: Examples for subsequence, longest–common–subsequence

sequence that is a subsequence of both A and B. For example, if $A = \langle c, a, d, b, r, z \rangle$ and $B = \langle a, s, b, z \rangle$, the longest common subsequence of A and B is $\langle a, b, z \rangle$.

Let $F[i,j]$ denote the length of the longest common subsequence of the first i elements of A and the first j elements of B. The objective of the LCS problem is to determine $F[n,m]$. The DP formulation for this problem expresses $F[i,j]$ in terms of $F[i-1,j-1, F[i,j-1]$, and $F[i-1,j]$ as follows:

$$F[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ F[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max\{F[i,j-1], F[i-1,j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Given sequences A and B, consider two pointers pointing to the start of the sequences. If the entries pointed to by the two pointers are identical, then they form components of the longest common subsequence.

Therefore, both pointers can be advanced to the next entry of the respective sequences and the length of the longest common subsequence can be incremented by one. If the entries are not identical then two situations arise: the longest common subsequence may be obtained from the longest subsequence of A and the sequence obtained by advancing the pointer to the next entry of B; or, the longest subsequence may be obtained from the longest subsequence of B and the sequence obtained by advancing the pointer to the next entry of A. Since we want to determine the longest subsequence, the maximum of these two must be selected.

The sequential implementation of this DP formulation computes the values in table F in row-major order. Since there is a constant amount of computation at each entry in the table, the overall complexity of this algorithm is $O(nm)$. This DP formulation is nonserial monadic, as illustrated in Fig.?? 9.5(a). Treating nodes along a diagonal as belonging to one level, each node depends on two subproblems at the preceding level and one subproblem two levels earlier. The formulation is monadic because a solution to any subproblem at a level is a function of only one of the solutions at preceding levels. (Note that, for the third, case in in the recursive definition both $F[ij-1]$ and $F[i-1]$ are possible solutions to $F[i,j]$, and the optimal solution to $F[i,j]$ is the maximum of the two.) Fig.?? shows that this problem has a very regular structure.

To simplify the discussion, we discuss parallel formulation only for the case in which $n = m$. Consider a parallel formulation of this algorithm on a CREW PRAM with n processors. Each processor P_i computes the i^{th} column of table F. Table entries are computed on a diagonal sweep from the top-left to the bottom-right corner. Since there are n processors, and each processor can access any entry in table F, the elements of each diagonal are computed in constant time (the diagonal can contain at most n elements). Since there are $2n-1$ such diagonals, the algorithm requires $O(n)$ iterations. Thus, the parallel run time is $O(n)$. The algorithm is cost-optimal, since its $O(n^2)$ processor-time product equals the sequential complexity.

This algorithm can be adapted to run on a linear array of n processors

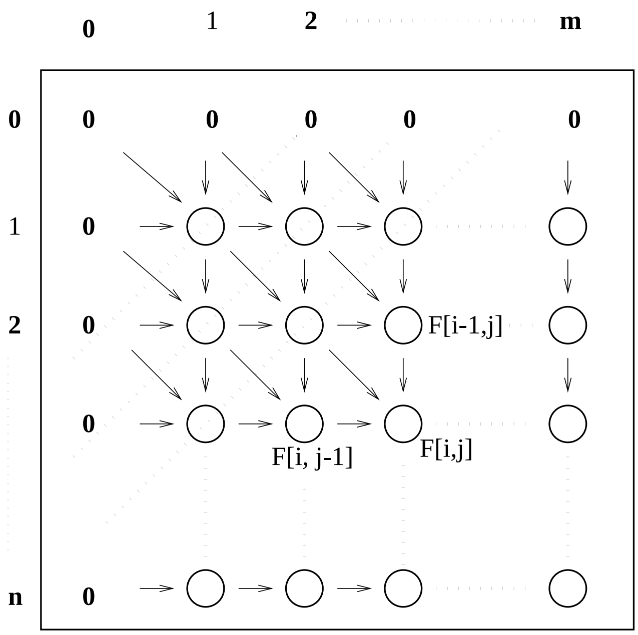

Figure 1.6:

by distributing table F among different processors. Processor P_i stores the $(i+1)^{st}$ column of the table. Entries in table F are assigned to processors as illustrated in Fig.??. When computing the value of $F[i,j]$, processor P_{j-1} may need either the value of $F[i-1, j-1]$ or the value of $F[i, j-1]$ from the processor to its left. It takes time $t_s + t_w$ to communicate a single word from a neighboring processor. To compute each entry in the table, a processor needs a single value from its immediate neighbor, followed by the actual computation, which takes time t_c. Since each processor computes a single entry on the diagonal, each iteration takes time $(t_s + t_w + t_c)$. The algorithm makes $(2m-1)$ diagonal sweeps (iterations) across the table; thus, the total parallel run time is

$$T_p = (2n-1)(t_s + t_w + t_c).$$

Since the sequential run time is $n^2 t_c$, the efficiency of this algorithm is

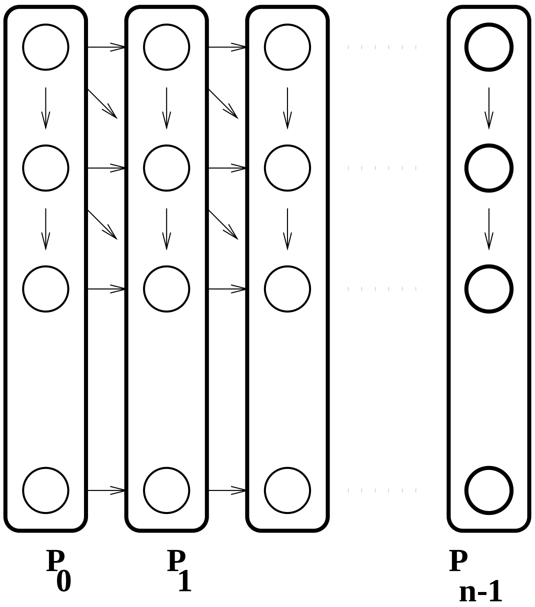

Figure 1.7:

$$E = \frac{n^2 t_c}{n(2n-1)(t_s + t_w + t_c)}.$$

] A careful examination of this expression reveals that it is not possible to obtain efficiencies above a certain threshold. To compute this threshold, assume it is possible to communicate values between processors instantaneously; that is, $t_s = t_w = 0$. In this case, the efficiency of the parallel algorithm is

$$E_{\max} = \frac{1}{2 - 1/n}.$$

Thus, the efficiency is bounded above by 0.5. This upper bound holds even if multiple columns are mapped to a processor. Higher efficiencies are possible using alternate mappings. Parallel formulations for architectures such as the mesh and hypercube only require embedding the ring into the target architecture.

Note that the basic characteristic that allows efficient parallel formulations of this algorithm is that table F can be partitioned so computing each element requires data only from neighboring processors. In other words, the algorithm exhibits locality of data access.

1.4 Serial Polyadic DP Formulations

In the Graph Algorithms chapter we have investigated the Floyd's algorithm for determining the shortest paths between all pairs of nodes in a graph. To give an example to serial polyadic DP formulations we reconsider the problem For basic notation and terminology we refer to the Graph Algorithms chapter.

1.4.1 DP formulation of the Floyd's All-Pairs Shortest-Paths Algorithm

Let G a weighted graph, which consists of a set of nodes V and a set of edges E. An edge from node i to node j in E has a weight c_i, j. Floyd's algorithm determines the cost $d_{i,j}$ of the shortest path between each pair of nodes (i, j) in V. The cost of a path is the sum of the weights of the edge in the path (see Fig.??).

For DP formulation we use a several technique to define the recursive equations like in the previous examples: we consider the $V^k = \{v_0, v_1, \ldots, v_{k-1}\}$ subset of the node set V, and we will stepwise add the next nodes till we reach the entire node set V.

Let $d^k_{i,j}$ be the minimum cost of a path from node i to node j, using only nodes V^k. The functional equation of the DP formulation for this problem is

$$d^k_{i,j} = \begin{cases} c_{i,j} & k = 0 \\ \min\{d^{k-1}_{i,j}, (d^{k-1}_{i,k}, +d^{k-1}_{k,j})\} & 0 \leq k \leq n-1 \end{cases}$$

Since $d^n_{i,j}$ is the shortest path from node i to node j using all n nodes, it is also the cost of the overall shortest path between nodes i and j.

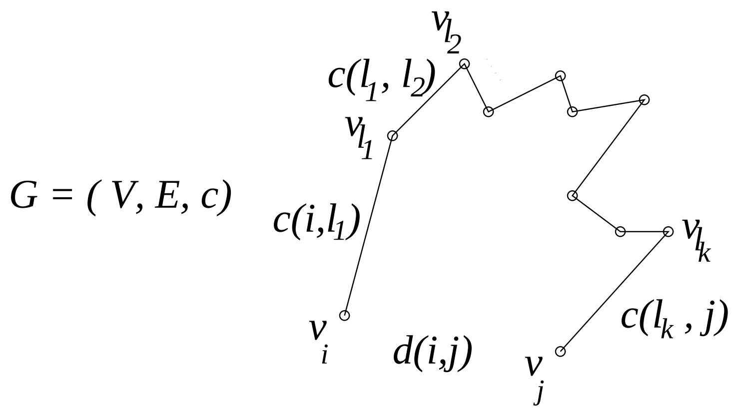

Figure 1.8: All-pairs shortest path

The sequential formulation of this algorithm requires n iterations, and each iteration requires $O(n^2)$ time. Thus, the overall run time of the sequential algorithm is $O(n^3)$.

The equation above is a serial polyadic formulation. Nodes $d^k_{i,j}$ can be partitioned into n levels, one for each value of k. Elements at level $k+1$ depend only on element at level k. Hence, the formulation is serial. The formulation is polyadic since one of the solution to $d^k_{i,j}$ requires a composition of solutions to two subproblems $d^{k-1}_{i,k}$ and $d^{k-1}_{k,j}$ from the previous level. Furthermore, the dependencies between levels are sparse because the computation of each element in $d^{k+!}_{i,j}$ requires only three results from the preceding level (out of n^2).

A simple CREW PRAM formulation of this algorithm uses n^2 processors. Processors are organized into a two-dimensional array in which processor $P_{i,j}$ computes the value of $d^k_{i,j}$ for $k = 1, 2, \ldots, n$. In each iteration k, processor $P_{i,j}$ requires the values $d^{k-1}_{i,j}$, $d^{k-1}_{i,k}$, and $d^{k-1}_{k,j}$. Given these values, it computes the value of $d^k_{i,j}$ in constant time. Therefore, the PRAM formulation has a parallel run time of $O(n)$. This formulation is cost-optimal because its processor-time product is the same as the sequential run time of $O(n^3)$. This algorithm can be adapted to practical architectures like the hypercube to yield efficient parallel formulations.

As with serial monadic formulations, data locality is of prime importance in serial polyadic formulations since many such formulations have sparse connectivity between levels.

1.5 Nonserial Polyadic DP Formulations

In nonserial polyadic DP formulations, in addition to processing subproblems at a level in parallel, computation can also be pipelined to increase efficiency. We illustrate this with the optimal matrix-parenthesization problem.

1.5.1 The Optimal Matrix-Parenthesization Problem

Consider the problem of multiplying n matrices, $A_1, A_2, \ldots, A_n$, where each A_i is a matrix with r_{i-1} rows and r_i columns. (See Fig.??.) The order in which the matrices are multiplied has a significant impact on the total number of operations required to evaluate the product. For example, consider three matrices A_1, A_2, and A_3 of dimensions $10 \times 20, 20 \times 30$, and 30×40, respectively. The product of these matrices can be computed as $(A_1 \times A_2) \times A_3$ or as $A_1 \times (A_2 \times A_3)$. In $(A_1 \times A_2) \times A_3$, computing $(A_1 \times A_2)$ requires $10 \times 20 \times 30$ operations and yields a matrix of dimensions 10×30. Multiplying this by A_3 requires $10 \times 30 \times 40$ additional operations. Therefore the total number of operations is $10 \times 20 \times 30 + 10 \times 30 \times 40 = 18,000$. Similarly, computing $A_1 \times (A_2 \times A_3)$ requires $20 \times 30 \times 40 + 10 \times 20 \times 40 = 32,000$ operations. Clearly, the first parethesization is desirable. The objective of the parenthesization problem is to determine a parenthesization that minimizes the number of operations. Enumerating all possible parenthesizations feasible since there are exponentially many of them.

In order to formulise the recursive equation let we define $C[i, j]$ as the optimal cost of multiplying the matrices $A_i, \ldots, A_j$. But that chain of matrices can be expressed as a product of two subchains

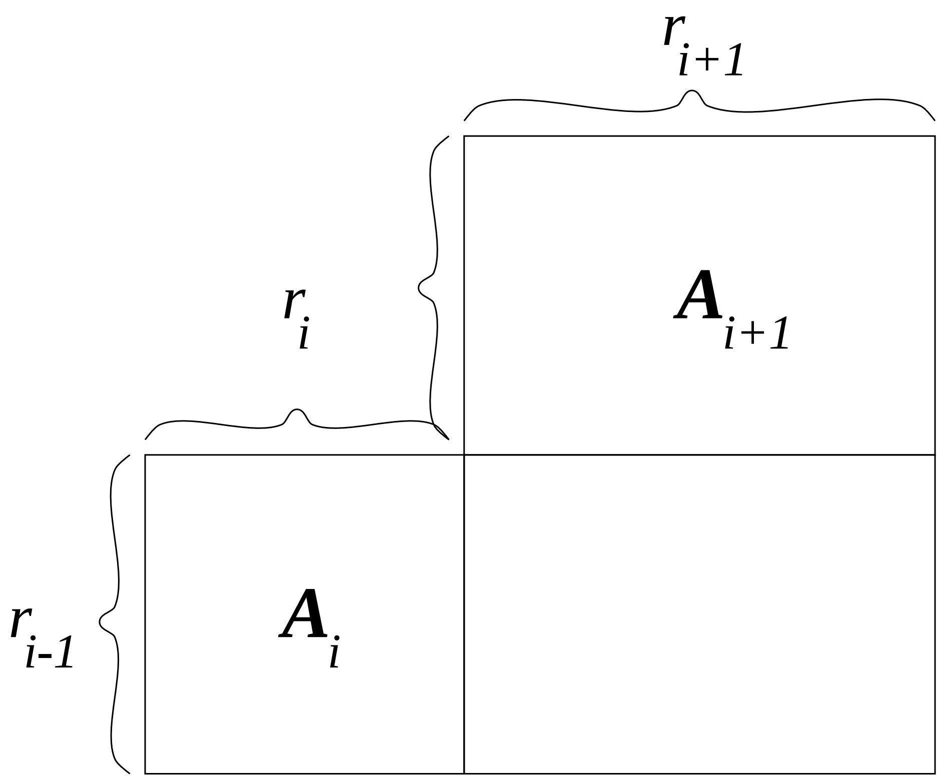

Figure 1.9: Multiplying matrices

i.e. $A_i, A_{i+1}, \ldots, A_k$ and $A_{k+1}, \ldots, A_j$. If we multiply the matrices $A_i, \ldots, A_k$ we got a matrix with dimension $r_{i-1} \times r_k$, and multiplying the rest, $A_{k+1}, \ldots, A_j$ it results a matrix of dimension $r_k \times r_j$. Now, if we multiply the resultant matrices we have to perform $r_{i-1} \times r_k \times r_j$ operation. This means if we parenthesize in the form $(A_i, \ldots, A_k A_{k+1}, \ldots, A_j)$ then the corresponding cost is $C[i,k]+C[k+1,j]+r_{i-1}r_kr_j$. Considering the optimal parenthesization of the matrices $A_i, \ldots, A_j$ there exists a k index which yield the optimal cost. In order to find it we have to compute every costs where $i \leq k < j$. Base on this we can set up the DP recurrence relation in the following form:

$$C[i,j] = \begin{cases} 0 & j = i, 0 < i \leq n \\ \min_{i \leq k < j}\{C[i,k] + C[k+1,j] + r_{i-1}r_kr_j\} & 1 \leq i < j \leq n \end{cases}$$

The problem is to find the $C[1,n]$. We set up a table containing the $C[i,j]$ values. To fill the table we consider first the matrix parenthetisation problem of length 1 then length 2 and so on. Based on the already filled cells in the table we are able to fill new cells by using the recirsive equation above. The table is depicted on the Fig.??. Entries in the diagonal l corresponds to the cost of multiplying matrices of length $l+1$. Let we denote a computation time of one term by t_c.

In computing the cost of the optimal parenthesization sequence the algorithm computes $(n-1)$ chains of length 2. This takes $(n-1)t_c$ computation time. The next step is to compute the chains having length 3. This takes $(n-2)3t_c$ time. At last we have to compute a chain of length n. As a result we set the followin formula:

$$T_S = t_c((n-1) + 2(n-2) + \ldots + (n-1)1) = \sum_i = 1^{n-1}(n-i)it_c$$

which is approx. equal $(n^3/6)t_c$.

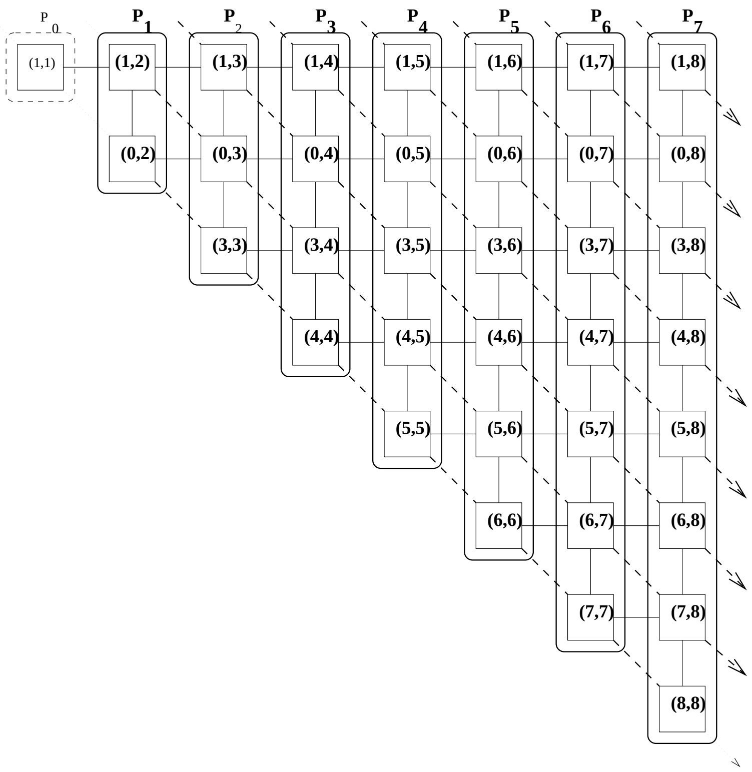
P0
P1
P2
P3
P4
P5
P6
P7
(1,1)
(1,2)
(1,3)
(1,4)
(1,5)
(1,6)
(1,7)
(1,8)
(0,2)
(0,3)
(0,4)
(0,5)
(0,6)
(0,7)
(0,8)
(3,3)
(3,4)
(3,5)
(3,6)
(3,7)
(3,8)
(4,4)
(4,5)
(4,6)
(4,7)
(4,8)
(5,5)
(5,6)
(5,7)
(5,8)
(6,6)
(6,7)
(6,8)
(7,7)
(7,8)
(8,8)

Figure 1.10: