Dynamic Programming Recurrences

1 Summary

Longest Common Subsequence (Section 2). Given two sequences $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$, we define $c[i, j]$ $(0 \leq i \leq m, 0 \leq j \leq n)$ to be the length of an LCS of $\langle x_1, x_2, \ldots x_i \rangle$ and $\langle y_1, y_2, \ldots y_j \rangle$. Then $c[m, n]$ is the length of an LCS of X and Y, and can be computed using the following recurrence relation (see, e.g., [5]):

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

Pairwise Sequence Alignment with Affine Gap Costs (Section 3). Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ over a finite alphabet Σ, a *gap introduction cost* g_i, *gap extension cost* g_e, and a mismatch cost $s(a, b)$ for each pair $a, b \in \Sigma$, the minimum cost of aligning X and Y by solving the following dynamic programming recurrences [9].

$$D[i,j] = \begin{cases} G[0,j] & \text{if } i = 0 \wedge j > 0, \\ \min \begin{Bmatrix} D[i-1,j], \\ G[i-1,j] + g_i \end{Bmatrix} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases} \quad I[i,j] = \begin{cases} G[i,0] & \text{if } i > 0 \wedge j = 0, \\ \min \begin{Bmatrix} I[i,j-1], \\ G[i,j-1] + g_i \end{Bmatrix} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases}$$

$$G[i,j] = \begin{cases} 0 & \text{if } i = 0 \wedge j = 0, \\ g_i + g_e \times j & \text{if } i = 0 \wedge j > 0, \\ g_i + g_e \times i & \text{if } i > 0 \wedge j = 0, \\ \min \begin{Bmatrix} D[i,j], \\ I[i,j], \\ G[i-1,j-1] + s(x_i, y_j) \end{Bmatrix} & \text{if } i > 0 \wedge j > 0. \end{cases} \tag{2}$$

The optimal alignment cost is $\min\{G[m,n], D[m,n], I[m,n]\}$.

The Gap Problem (Section 4). Let $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ be two given strings, and w and w' be two given functions, where $w(p, q)$ $(0 \leq p < q \leq m)$ is the cost of deleting $x_{p+1} \ldots x_q$ from X, and $w'(p, q)$ $(0 \leq p < q \leq n)$ is the cost of inserting $y_{p+1} \ldots y_q$ into X. The substitution function $S(x_i, y_j)$ is the same as that of the standard edit distance problem.

Let $G[i, j]$ denote the minimum cost of transforming $X_i = x_1 x_2 \ldots x_i$ into $Y_j = y_1 y_2 \ldots y_j$ (where $0 \leq i \leq m$ and $0 \leq j \leq n$) under this general setting. Then

$$G[i,j] = \begin{cases} 0 & \text{if } i = j = 0, \\ w(0,j) & \text{if } i = 0,\ 1 \leq j \leq n, \\ w'(0,i) & \text{if } j = 0,\ 1 \leq i \leq m, \\ \min \begin{Bmatrix} G[i-1,j-1] + S(x_i,\ y_j), \\ \min_{0 \leq q < j} \{\ G[i,q] + w(q,j)\ \}, \\ \min_{0 \leq p < i} \{\ G[p,j] + w'(p,i)\ \} \end{Bmatrix} & \text{if } i, j > 0. \end{cases} \tag{3}$$

The Least Weight Subsequence Problem (Section 5). This problem [8, 10] is defined by the following recurrence, and can be viewed as a one dimensional version of the gap problem:

$$D[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min_{0 \le i < j} (D[i] + w(i,j)) & \text{otherwise;} \end{cases} \tag{4}$$

where w is a real-valued function.

Protein Accordion Folding (Section 8). This problem assumes that a protein sequence $\mathcal{P}[1:n]$ is folded in an accordion fold into a 2D square lattice in such a way that the number of pairs of hydrophobic amino acids that are next to each other in the grid (vertically or horizontally) without being next to each other in the protein sequence is maximized. We have a precomputed array $\mathrm{SOF}[1:n, 1:n]$ such that for all $1 \le i < j < k-1 < n$, $\mathrm{SOF}[j+1, \min\{k, 2j-i+1\}]$. gives the number of aligned hydrophobic amino acids when the protein segment $\mathcal{P}[i:k]$ is folded only once at indices $(j, j+1)$.

The recurrence below shows how to compute the optimal *accordion score* of the protein segment $\mathcal{P}[i:j]$. The optimal score for the entire sequence is given by $\max_{1<j\le n}\{\textsc{Score}[1,j]\}$.

$$\textsc{Score}[i,j] = \begin{cases} 0 & \text{if } j \ge n-1, \\ \max_{j+1<k\le n}\{\mathrm{SOF}[j+1, \min\{k, 2j-i+1\}] + \textsc{Score}[j+1,k]\} & \text{otherwise.} \end{cases} \tag{5}$$

Floyd-Warshall's All-Pairs Shortest Paths (Section 7). Consider a directed graph $\mathcal{G} = (V,E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and each edge (v_i, v_j) is labeled by an element $l(v_i, v_j)$ of some closed semiring $(S, \oplus, \odot, 0, 1)$. For $i, j \in [1,n]$ and $k \in [0,n]$, let $d[i,j,k]$ denote cost of the smallest cost path from v_i to v_j with no intermediate vertex higher than v_k. Then $d[i,j,n]$ is the cost of the shortest path from v_i to v_j. The following recurrence computes all $d[i,j,k]$.

$$d[i,j,k] = \begin{cases} 1 & \text{if } k = 0 \text{ and } i = j, \\ l(v_i, v_j) & \text{if } k = 0 \text{ and } i \ne j, \\ d[i,j,k-1] \oplus (d[i,k,k-1] \odot d[k,j,k-1]) & \text{otherwise.} \end{cases} \tag{6}$$

Floyd-Warshall's APSP performs computations over a particular closed semiring $(\Re, \min, +, +\infty, 0)$.

The Parenthesis Problem (Section 6). The *parenthesis problem* [8] is defined by the following recurrence relation:

$$c[i,j] = \begin{cases} x_j & \text{if } 0 \le i = j-1 < n, \\ \min_{i<k<j} \left\{ \begin{matrix} (c[i,k] + c[k,j]) \\ + w(i,k,j) \end{matrix} \right\} & \text{if } 0 \le i < j-1 < n; \end{cases} \tag{7}$$

where x_j's are assumed to be given for $j \in [1,n]$. We also assume that $w(\cdot,\cdot,\cdot)$ is a function that can be computed in-core without additional memory accesses.

Basic RNA Secondary Structure Prediction without Pseudoknots (Section 9). Given an RNA string $X = x_1 x_2 \ldots x_n$, the *basic RNA secondary structure prediction* problem asks for an RNA secondary structure (w/o pseudoknots) with the maximum number of base pairs. Let $R[i,j]$ denote the maximum number of base pairs in a secondary structure formed by the RNA substrand $x_i x_{i+1} \ldots x_j$. Then the following recurrence can be used to compute $R[i,j]$ for all $i, j \in [1,n]$ [11].

$$R[i,j] = \begin{cases} 0 & \text{if } i \geq j-4, \\ \max\left\{ R[i,j-1],\ \max\limits_{\substack{i<k<j-1, \\ \{x_k, x_j\} \text{ is a base pair}}} \left\{ \begin{array}{c} 1 \ + \ R[i,k-1] \\ + R[k+1,j-1] \end{array} \right\} \right\} & \text{otherwise.} \end{cases} \tag{8}$$

Cocke-Younger-Kasami (CYK) Algorithm (Section 11). We are given a string $X = \langle x_1, x_2, \ldots, x_n \rangle$ and a context-free grammar $G = (V, \sum, R, S)$, where V is a set of variables (or non-terminal symbols), $\sum$ is a finite set of terminal symbols, $R = \{R_1, R_2, \ldots, R_t\} : V \to (V \cup \sum)^*$ is a finite set of rules, and S is a start variable chosen from V. We set $P[i,j,c]$ to ***true*** provided substring $X_{ij} = x_i x_{i+1} \ldots x_{i+j-1}$ can be generated from rule $R_c \in R$, and to ***false*** otherwise. Then

$$P[i,j,c] = \begin{cases} \textbf{\textit{true}} & \text{if } j = 1 \text{ and } R_c \to x_i, \\ \textbf{\textit{true}} & \text{if } P[i,k,a] = P[i+k, j-k, b] = \textbf{\textit{true}}, k \in [1, i-1] \text{ and } R_c \to R_a R_b, \\ \textbf{\textit{false}} & \text{otherwise.} \end{cases} \tag{9}$$

0-1 Knapsack Problem (Section 12). We are given a knapsack of capacity W, a set of n items $\{x_1, x_2, \ldots, x_n\}$, where item x_i has a value v_i and weight w_i. Let $K[i,j]$ denote the maximum value of a subset of $\{x_1, x_2, \ldots, x_i\}$ that is less than or equal to j. Then

$$K[i,j] = \begin{cases} 0 & \text{if } j = 0, \\ K[i-1, j] & \text{if } w_i > w, \\ \max(K[i-1,j], K[i-1, j-w_i] + v_i) & \text{otherwise.} \end{cases} \tag{10}$$

Viterbi Algorithm (Section 13). We are given an observation space $O = \{o_1, o_2, \ldots, o_N\}$, state space $S = \{s_1, s_2, \ldots, s_K\}$, observations $Y = \{y_1, y_2, \ldots, y_T\}$, transition matrix A, where $A[i,j]$ is the transition probability of transiting from s_i to s_j, emission matrix B, where $B[i,j]$ is the probability of observing o_j from s_i, initial probability array I, where $I[i]$ is the probability that $x_i = s_i$. Then,

$$P[i,j] = \begin{cases} I[i] \times B[i, y_1] & \text{if } j = 1, \\ \max_{k \in [1,K]}(P[k, j-1] \cdot A[k,i] \cdot B[i, y_j]) & \text{otherwise.} \end{cases} \tag{11}$$

Eggs Problem (Section 14). A building has n floors and we are given k eggs. There might be a threshold floor in the building from and below which when an egg is dropped, the egg does not break, and above which when the egg is dropped, the egg breaks. Let $D[i][j]$ denote the minimum number of drops for first i floors and using j eggs. Then,

$$D[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \min_{k \in [1,i]}(\max(D[k-1, j-1], D[i-k, j])) & \text{otherwise.} \end{cases} \tag{12}$$

2 Longest common subsequence

A sequence $Z = \langle z_1, z_2, \ldots z_k \rangle$ is called a *subsequence* of another sequence $X = \langle x_1, x_2, \ldots x_m \rangle$ if there exists a strictly increasing function $f : [1,2,\ldots,k] \to [1,2,\ldots,m]$ such that for all $i \in [1,k]$, $z_i = x_{f(i)}$. A sequence Z is a *common subsequence* of sequences X and Y if Z is a subsequence of both X and Y. In the *Longest Common Subsequence* (LCS) problem we are given two sequences X and Y, and we need to find a maximum-length common subsequence of X and Y.

Given two sequences $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$, we define $c[i,j]$ $(0 \le i \le m, 0 \le j \le n)$ to be the length of an LCS of $\langle x_1, x_2, \ldots x_i \rangle$ and $\langle y_1, y_2, \ldots y_j \rangle$. Then $c[m,n]$ is the length of an LCS of X and Y, and can be computed using the following recurrence relation (see, e.g., [5]):

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max\{c[i,j-1], c[i-1,j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{13}$$

The classic dynamic programming solution to the LCS problem is based on this recurrence relation, and computes the entries of $c[0\ldots m, 0\ldots n]$ in row-major order in $\Theta(mn)$ time and space.

3 Pairwise sequence alignment with affine gap costs

Sequence alignment plays a central role in biological sequence comparison, and can reveal important relationships among organisms. Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ over a finite alphabet Σ, an *alignment* of X and Y is a matching M of sets $\{1,2,\ldots,m\}$ and $\{1,2,\ldots,n\}$ such that if $(i,j),(i',j') \in M$ and $i < i'$ hold then $j < j'$ must also hold. The i-th letter of X or Y is said to be in a *gap* if it does not appear in any pair in M. Given a *gap penalty* g and a mismatch cost $s(a,b)$ for each pair $a,b \in \Sigma$, the *basic (global) pairwise sequence alignment problem* asks for a matching M_{opt} for which $(m+n-|M_{opt}|) \times g + \sum_{(a,b)\in M_{opt}} s(a,b)$ is minimized.

The formulation of the basic sequence alignment problem favors a large number of small gaps while real biological processes favor the opposite. The alignment can be made more realistic by using an *affine gap penalty* which has two parameters: a *gap introduction cost* g_i and a *gap extension cost* g_e. A run of t gaps incurs a total cost of $g_i + g_e \times t$. Such an alignment with minimum cost can be found by solving the following dynamic programming recurrences [9] (observe the similarity between the recurrence for G and the LCS recurrence [5]).

$$D[i,j] = \begin{cases} G[0,j] & \text{if } i = 0 \wedge j > 0, \\ \min\left\{\begin{matrix} D[i-1,j], \\ G[i-1,j]+g_i \end{matrix}\right\} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases} \quad I[i,j] = \begin{cases} G[i,0] & \text{if } i > 0 \wedge j = 0, \\ \min\left\{\begin{matrix} I[i,j-1], \\ G[i,j-1]+g_i \end{matrix}\right\} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases}$$

$$G[i,j] = \begin{cases} 0 & \text{if } i = 0 \wedge j = 0, \\ g_i + g_e \times j & \text{if } i = 0 \wedge j > 0, \\ g_i + g_e \times i & \text{if } i > 0 \wedge j = 0, \\ \min\left\{\begin{matrix} D[i,j], \\ I[i,j], \\ G[i-1,j-1]+s(x_i,y_j) \end{matrix}\right\} & \text{if } i > 0 \wedge j > 0. \end{cases} \tag{14}$$

The optimal alignment cost is $\min\{G[m,n], D[m,n], I[m,n]\}$ and an optimal alignment can be traced back from the smallest of $G[m,n]$, $D[m,n]$ and $I[m,n]$.

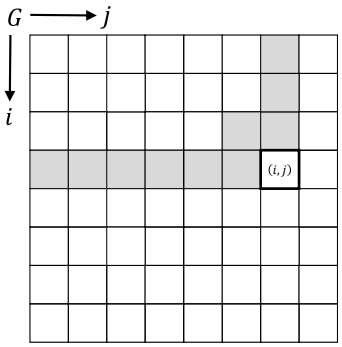

Figure 1: Dependency structure of the gap DP: cell (i, j) depends on the grey cells.

4 The gap problem

The *gap* problem [7,8,15] is a generalization of the edit distance problem that arises in molecular biology, geology, and speech recognition. When transforming a string $X = x_1x_2\ldots x_m$ into another string $Y = y_1y_2\ldots y_n$, a sequence of consecutive deletes corresponds to a gap in X, and a sequence of consecutive inserts corresponds to a gap in Y. In many applications the cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap. In order to handle this general case two new cost functions w and w' are defined, where $w(p,q)$ $(0 \le p < q \le m)$ is the cost of deleting $x_{p+1}\ldots x_q$ from X, and $w'(p,q)$ $(0 \le p < q \le n)$ is the cost of inserting $y_{p+1}\ldots y_q$ into X. The substitution function $S(x_i, y_j)$ is the same as that of the standard edit distance problem.

Let $G[i,j]$ denote the minimum cost of transforming $X_i = x_1x_2\ldots x_i$ into $Y_j = y_1y_2\ldots y_j$ (where $0 \le i \le m$ and $0 \le j \le n$) under this general setting. Then

$$G[i,j] = \begin{cases} 0 & \text{if } i = j = 0, \\ w(0,j) & \text{if } i = 0,\ 1 \le j \le n, \\ w'(0,i) & \text{if } j = 0,\ 1 \le i \le m, \\ \min \left\{ \begin{array}{c} G[i-1,j-1] + S(x_i,\ y_j), \\ \min_{0 \le q < j} \{\ G[i,q] + w(q,j)\ \}, \\ \min_{0 \le p < i} \{\ G[p,j] + w'(p,i)\ \} \end{array} \right\} & \text{if } i,j > 0. \end{cases} \tag{15}$$

Assuming $m = n$, this problem can be solved in $O(n^3)$ time using $O(n^2)$ space [7].

In the rest of this section we will assume for simplicity that $m = n = 2^t$ for some integer $t \ge 0$.

In Figures 2, 3 and 4 a cache-oblivious parallel algorithm is given for solving the gap problem. The algorithm consists of three recursive divide-and-conquer functions named A_{gap}, B_{gap} and C_{gap}. Figure 1 shows the dependency structure of the DP.

Cache Complexity on Serial Machines. For $f \in \{A,B,C\}$, let $Q_f(n)$ denote the cache complexity of f_{gap} on sequences of length n. Then

$$Q_A(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \le \gamma_A M, \\ 4Q_A\left(\frac{n}{2}\right) + 2Q_B\left(\frac{n}{2}\right) + 2Q_C\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

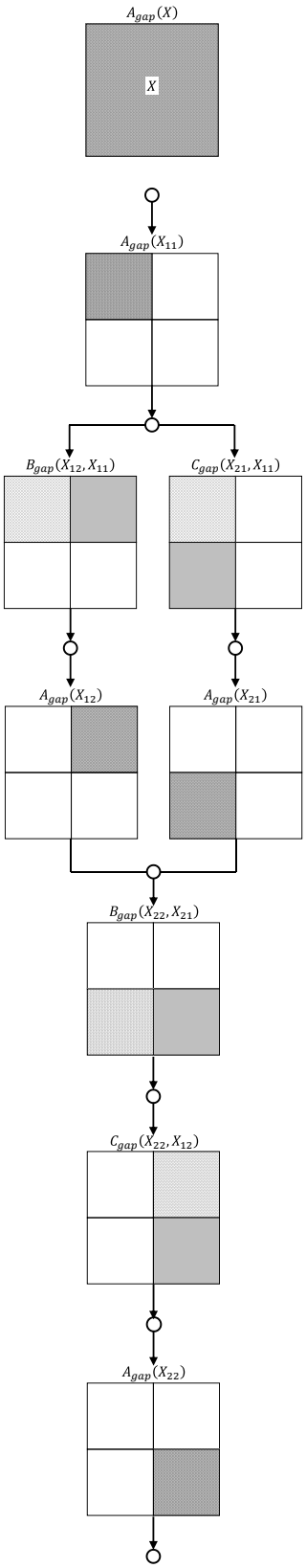

Figure 2: A cache-oblivious parallel divide-and-conquer algorithm for solving the gap problem. The initial call to the function is $A_{gap}(G)$. Functions B_{gap} and C_{gap} are given in Figures 3 and 4, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

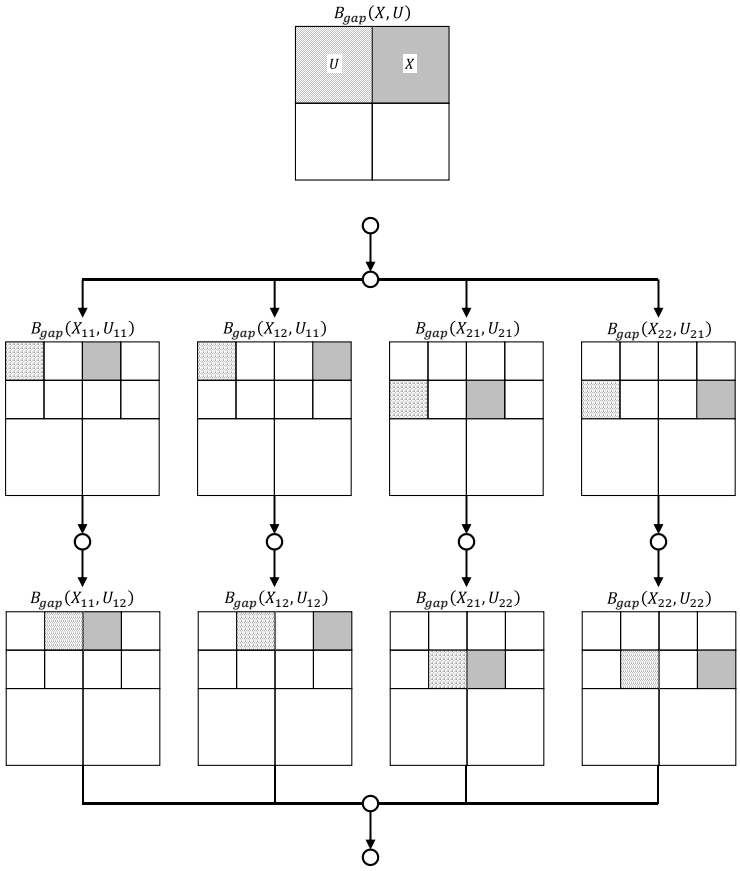

Figure 3: Function B_{gap} is invoked by function A_{gap} given in Figure 2 which is a cache-oblivious parallel divide-and-conquer algorithm for solving the gap problem. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

$$Q_B(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_B M, \\ 8Q_B\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases} \qquad Q_C(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_C M, \\ 8Q_C\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

where, γ_A, γ_B and γ_C are suitable constants. Solving, $Q_A(n) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$.

Span. For $f \in \{A,B,C\}$, let $T_f(n)$ denote the span of f_{gap} on sequences of length n. Then

$$T_A(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 3T_A\left(\frac{n}{2}\right) + \max\left\{T_B\left(\frac{n}{2}\right), T_C\left(\frac{n}{2}\right)\right\} + T_B\left(\frac{n}{2}\right) + T_C\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

$$T_B(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_B\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad T_C(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_C\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

Solving, $T_A(n) = O\left(n^{\log_2 3}\right)$.

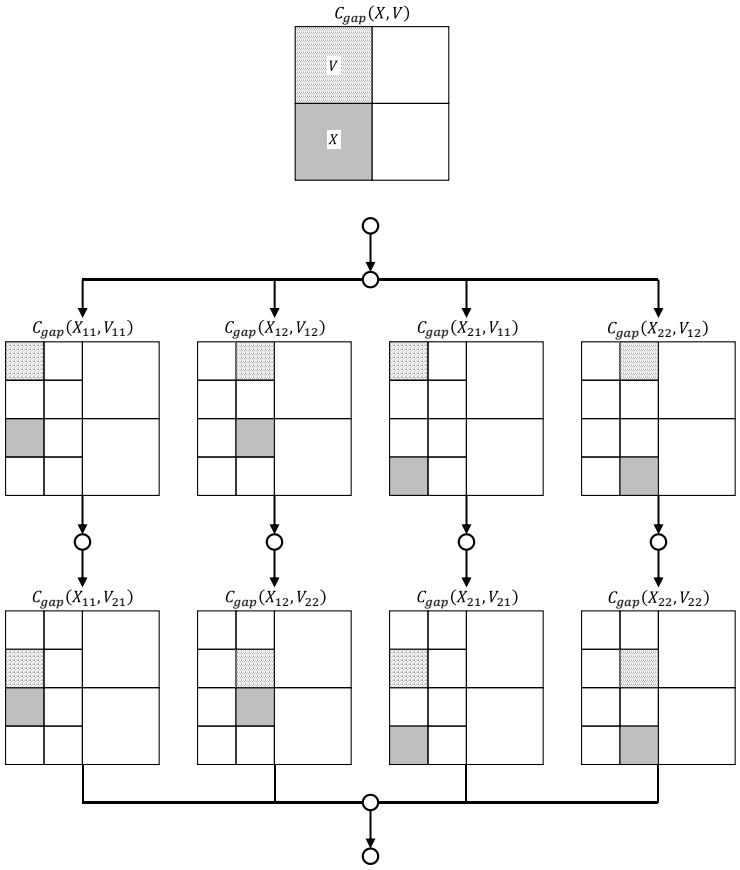

Figure 4: Function C_{gap} is invoked by function A_{gap} given in Figure 2 which is a cache-oblivious parallel divide-and-conquer algorithm for solving the gap problem. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

Cache Complexity on a Parallel Machine with Private Caches. Consider running the algorithm on p processors each with a private cache of size M and block size B. Then w.h.p. in n, the total number of cache misses incurred by the algorithm under the work-stealing scheduler is $\leq Q_A(n) + O\left(pT_A(n) \times \frac{M}{B}\right) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}} + p\left(\frac{M}{B}\right) n^{\log_2 3}\right)$.

5 The least weight subsequence problem

The *least weight subsequence* problem [8, 10] which is defined by the following dynamic programming recurrence, can be viewed as a one dimensional version of the gap problem:

$$D[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min_{0 \leq i < j} (D[i] + w(i,j)) & \text{otherwise;} \end{cases} \tag{16}$$

where w is a real-valued function. This problem can be solved cache-obliviously in $O(n^2)$ time, using $O(n)$ space and incurring $O\left(\frac{n^2}{BM}\right)$ cache misses using a one dimensional version of RECURSIVE-GAP, provided

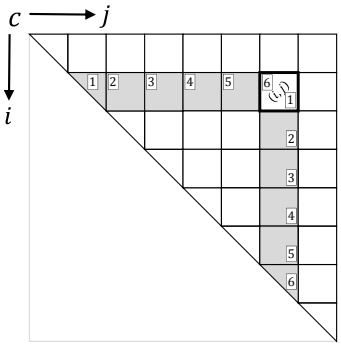

Figure 5: Dependency structure of the parenthesis DP: cell (i, j) depends on the grey cells. Cell numbered k on the horizontal grey line is paired with cell numbered k on the vertical grey line during the update of cell (i, j).

either $w(i, j)$ is a function of $j - i$, or it can be generated on the fly in constant time without incurring any additional cache misses.

This problem arises in optimum paragraph formation and in finding a minimum height B-tree [8].

6 The parenthesis problem

The *parenthesis problem* [8] is defined by the following recurrence relation:

$$c[i,j] = \begin{cases} x_j & \text{if } 0 \le i = j-1 < n, \\ \min_{i<k<j} \left\{ \begin{array}{c} (c[i,k]+c[k,j]) \\ +w(i,k,j) \end{array} \right\} & \text{if } 0 \le i < j-1 < n; \end{cases} \tag{17}$$

where x_j's are assumed to be given for $j \in [1,n]$. We also assume that $w(\cdot,\cdot,\cdot)$ is a function that can be computed in-core without additional memory accesses.

The class of problems defined by the recurrence relation above includes RNA secondary structure prediction, optimal matrix chain multiplication, construction of optimal binary search trees, and optimal polygon triangulation. A variant of this recurrence which does not include the $w(i,k,j)$ term and is defined as the *simple dynamic program*, was considered in [3].

As in [3], instead of recurrence 17 we will use the following slightly augmented version of 17 which will considerably simplify the recursive subdivision process in our cache-oblivious algorithm.

$$c[i,j] = \begin{cases} \infty & \text{if } 0 \le i = j \le n, \\ x_j & \text{if } 0 \le i = j-1 < n, \\ \min_{i\le k\le j} \left\{ \begin{array}{c} (c[i,k]+c[k,j]) \\ +w(i,k,j) \end{array} \right\} & \text{if } 0 \le i < j-1 < n; \end{cases} \tag{18}$$

where $w(i,k,j)$ is defined to be ∞ when $k = i$ or $k = j$. It is straight-forward to see that recurrences 17 and 18 are equivalent, i.e., they compute the same values for any given $c[i, j]$, $0 \le i < j-1 < n$.

In the rest of this section we will assume for simplicity that $n = 2^t - 1$ for some integer $t \ge 0$.

In Figures 6, 7 and 8 a cache-oblivious parallel algorithm is given for solving the parenthesis problem. The algorithm consists of three recursive divide-and-conquer functions named A_{par}, B_{par} and C_{par}. Figure 5 shows the dependency structure of the DP.

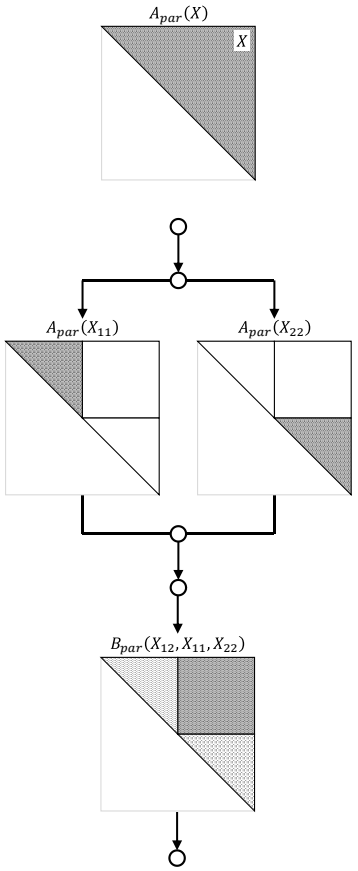

Figure 6: A cache-oblivious parallel divide-and-conquer algorithm for solving the parenthesis problem. The initial call to the function is $A_{par}(c)$. Functions B_{par} and C_{par} are given in Figures 7 and 8, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

Cache Complexity on Serial Machines. For $f \in \{A, B, C\}$, let $Q_f(n)$ denote the cache complexity of f_{par} on a matrix of size $n \times n$. Then

$$Q_A(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_A M, \\ 2Q_A\left(\frac{n}{2}\right) + Q_B\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

$$Q_B(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_B M, \\ 4\left(Q_B\left(\frac{n}{2}\right) + Q_C\left(\frac{n}{2}\right)\right) & \text{otherwise;} \end{cases} \qquad Q_C(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_C M, \\ 8Q_C\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

where, γ_A, γ_B and γ_C are suitable constants. Solving, $Q_A(n) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$.

Span. For $f \in \{A, B, C\}$, let $T_f(n)$ denote the span of f_{par} on a matrix of size $n \times n$. Then

$$T_A(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T_A\left(\frac{n}{2}\right) + T_B\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

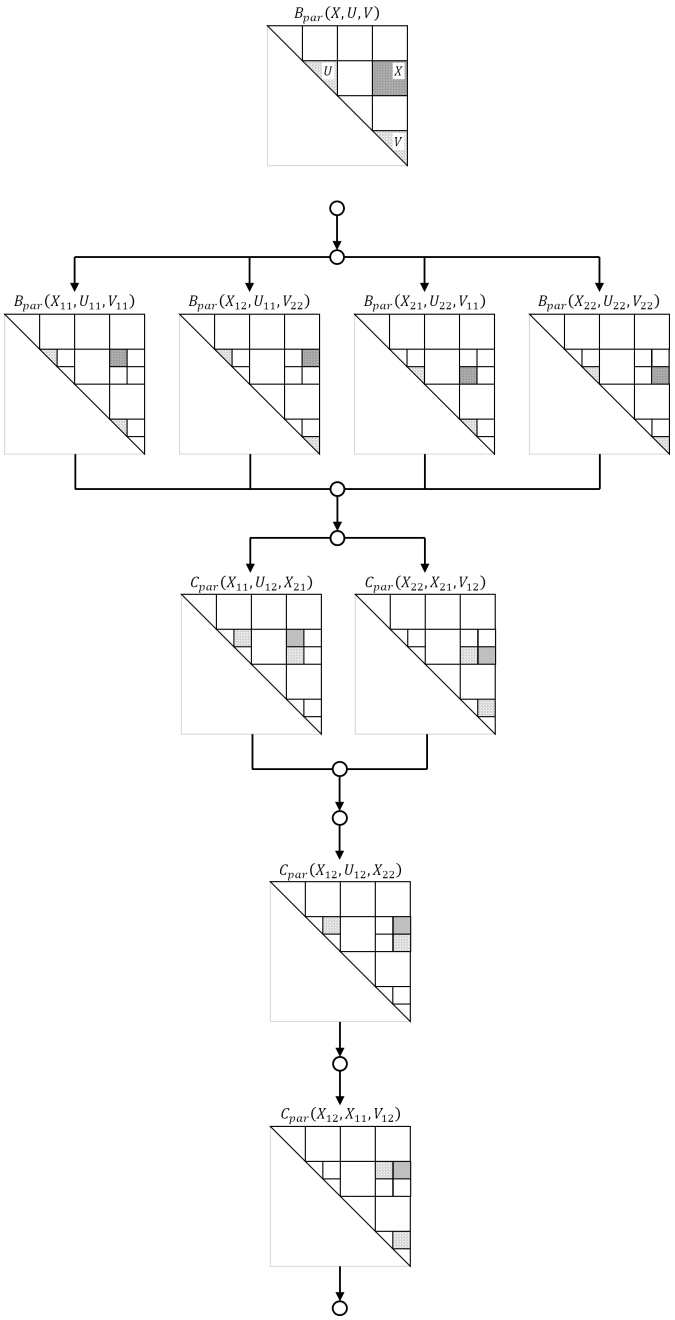

Figure 7: Function B_{par} is invoked by function A_{par} given in Figure 6 which is a cache-oblivious parallel divide-and-conquer algorithm for solving the parenthesis problem. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

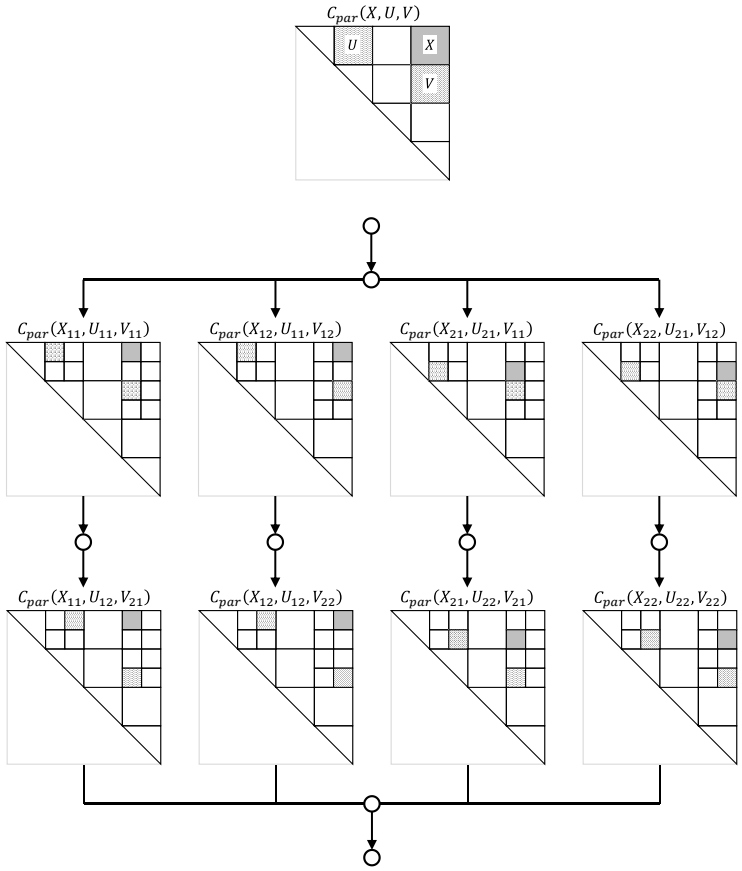

Figure 8: Function C_{par} is invoked by functions A_{par} and B_{par} given in Figures 6 and 7, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

$$T_B(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T_B\left(\frac{n}{2}\right) + 3T_C\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad T_C(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_C\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

Solving, $T_A(n) = O(n)$.

Cache Complexity on a Parallel Machine with Private Caches. Consider running the algorithm on p processors each with a private cache of size M and block size B. Then w.h.p. in n, the total number of cache misses incurred by the algorithm under the work-stealing scheduler is $\leq Q_A(n) + O\left(pT_A(n) \times \frac{M}{B}\right) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}} + p\left(\frac{M}{B}\right)n\right)$.

7 Floyd-Warshall's all-pairs shortest paths (APSP)

An algebraic structure known as a *closed semiring* [1] serves as a general framework for solving path problems in directed graphs. In [1], an algorithm is given for finding the set of all paths between each pair of vertices in a directed graph. Both Floyd-Warshall's algorithm for finding all-pairs shortest paths [6] and

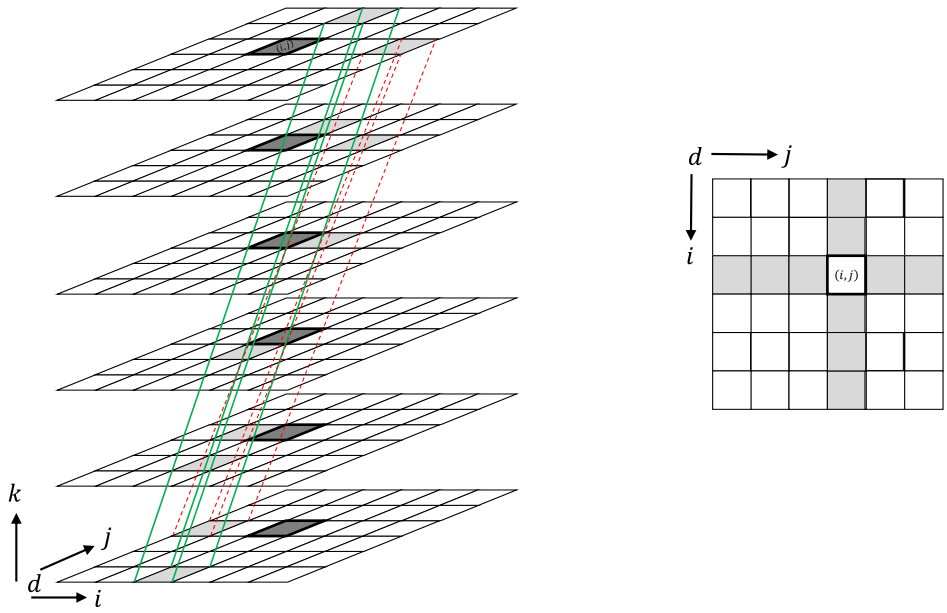

Figure 9: Dependency structure of the Floyd-Warshall DP: cell (i, j) depends on the grey cells.

Warshall's algorithm for finding transitive closures [14] are instantiations of this algorithm.

Consider a directed graph $\mathcal{G} = (V,E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and each edge (v_i, v_j) is labeled by an element $l(v_i, v_j)$ of some closed semiring $(S, \oplus, \odot, 0, 1)$. If $(v_i, v_j) \notin E$, $l(v_i, v_j)$ is assumed to have a value 0. The *path-cost* of a path is defined as the product ($\odot$) of the labels of the edges in the path, taken in order. The path-cost of a zero length path is 1. For each pair $v_i, v_j \in V$, $\delta[i, j]$ is defined to be the sum of the path-costs of all paths going from v_i to v_j. By convention, the sum over an empty set of paths is 0. Even if there are infinitely many paths between v_i and v_j (due to presense of cycles), $\delta[i, j]$ will still be well-defined due to the properties of a closed semiring.

For $i, j \in [1, n]$ and $k \in [0, n]$, let $d[i, j, k]$ denote cost of the smallest cost path from v_i to v_j with no intermediate vertex higher than v_k. Then $\delta[i, j] = d[i, j, n]$. The following recurrence computes all $d[i, j, k]$.

$$d[i,j,k] = \begin{cases} 1 & \text{if } k = 0 \text{ and } i = j, \\ l(v_i, v_j) & \text{if } k = 0 \text{ and } i \neq j, \\ d[i,j,k-1] \oplus (d[i,k,k-1] \odot d[k,j,k-1]) & \text{otherwise.} \end{cases} \tag{19}$$

The dependency structure of the DP is shown in Figure 9. The properties of the given semiring implies that $\delta[i, j]$ for all pairs of vertices $v_i, v_j \in V$ in only $O(n^2)$ space as shown in Figure 10. Floyd-Warshall's APSP performs computations over a particular closed semiring $(\Re, \min, +, +\infty, 0)$.

In the rest of this section we will assume for simplicity that $n = 2^t$ for some integer $t \geq 0$.

Figures 11 – 14 show a a cache-oblivious parallel implementation of Floyd-Warshall's APSP algorithm. The implementation consists of four recursive divide-and-conquer functions named A_{FW}, B_{FW}, C_{FW} and D_{FW}.

Cache Complexity on Serial Machines. For $f \in \{A, B, C, D\}$, let $Q_f(n)$ denote the cache complexity of f_{FW} on a matrix of size $n \times n$. Then

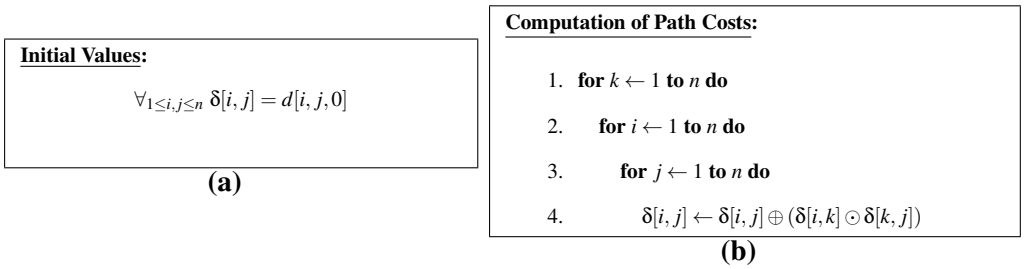

Figure 10: Computation of path costs over a closed semiring $(S, \oplus, \odot, 0, 1)$ in quadratic space: **(a)** Initialization of δ, **(b)** Computation of path costs.

$$Q_A(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \le \gamma_A M, \\ 2\left(Q_A\left(\frac{n}{2}\right) + Q_B\left(\frac{n}{2}\right) + Q_C\left(\frac{n}{2}\right) + Q_D\left(\frac{n}{2}\right)\right) & \text{otherwise;} \end{cases}$$

$$Q_B(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \le \gamma_B M, \\ 4\left(Q_B\left(\frac{n}{2}\right) + Q_D\left(\frac{n}{2}\right)\right) & \text{otherwise;} \end{cases} \qquad Q_C(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \le \gamma_C M, \\ 4\left(Q_C\left(\frac{n}{2}\right) + Q_D\left(\frac{n}{2}\right)\right) & \text{otherwise;} \end{cases}$$

$$Q_D(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \le \gamma_D M, \\ 8Q_D\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

where, γ_A, γ_B, γ_C and γ_D are suitable constants. Solving, $Q_A(n) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$.

Span. For $f \in \{A, B, C, D\}$, let $T_f(n)$ denote the span of f_{FW} on an input matrix of size $n \times n$. Then

$$T_A(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2\left(T_A\left(\frac{n}{2}\right) + \max\left\{T_B\left(\frac{n}{2}\right), T_C\left(\frac{n}{2}\right)\right\} + T_D\left(\frac{n}{2}\right)\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

$$T_B(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2\left(T_B\left(\frac{n}{2}\right) + T_D\left(\frac{n}{2}\right)\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad T_C(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2\left(T_C\left(\frac{n}{2}\right) + T_D\left(\frac{n}{2}\right)\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

$$T_D(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_D\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise.} \end{cases}$$

Solving, $T_A(n) = O\left(n \log^2 n\right)$.

Cache Complexity on a Parallel Machine with Private Caches. Consider running the algorithm on p processors each with a private cache of size M and block size B. Then w.h.p. in n, the total number of cache misses incurred by the algorithm under the work-stealing scheduler is $\le Q_A(n) + O\left(pT_A(n) \times \frac{M}{B}\right) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}} + p\left(\frac{M}{B}\right) n \log^2 n\right)$.

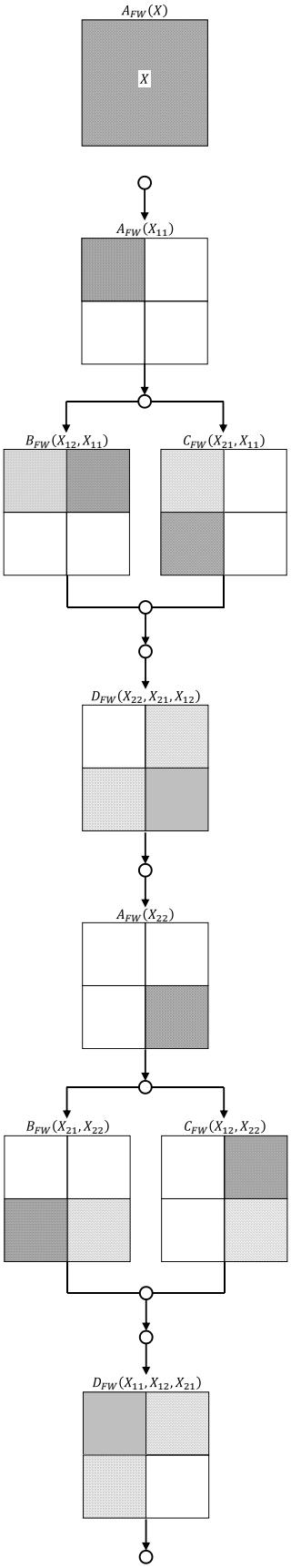

Figure 11: A cache-oblivious parallel divide-and-conquer implementation of Floyd-Warshall's APSP algorithm. The initial call to the function is $A_{FW}(\delta)$. Functions B_{FW}, C_{FW} and D_{FW} are given in Figures 12, 13 and 14, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

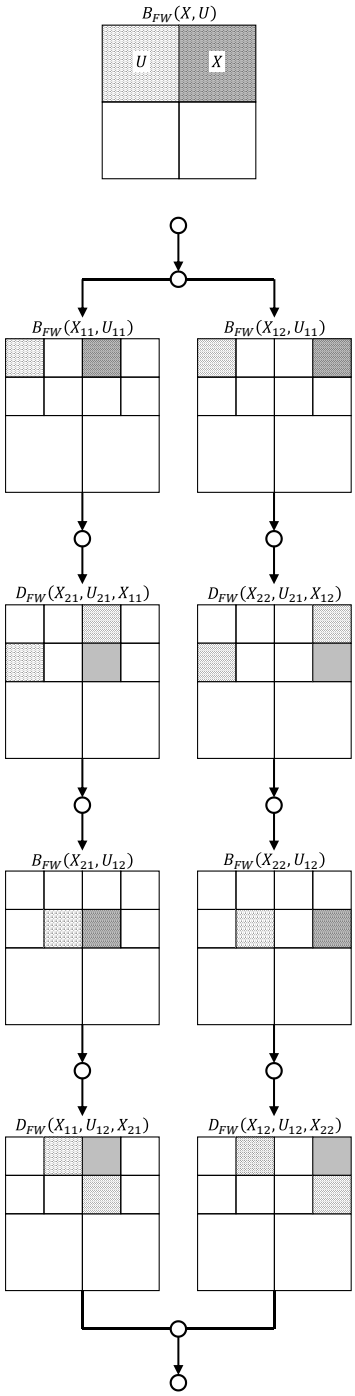

Figure 12: Function B_{FW} is invoked by function A_{FW} given in Figure 11. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

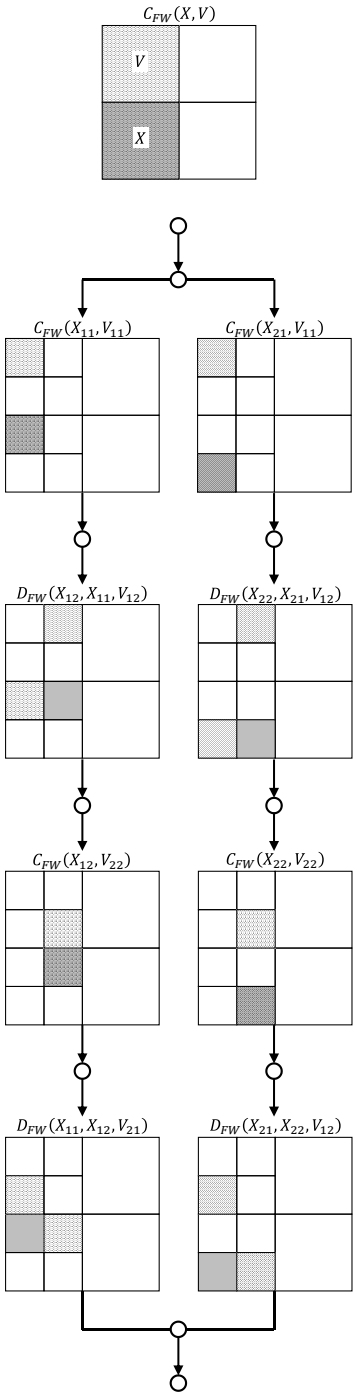

Figure 13: Function C_{FW} is invoked by function A_{FW} given in Figure 11. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

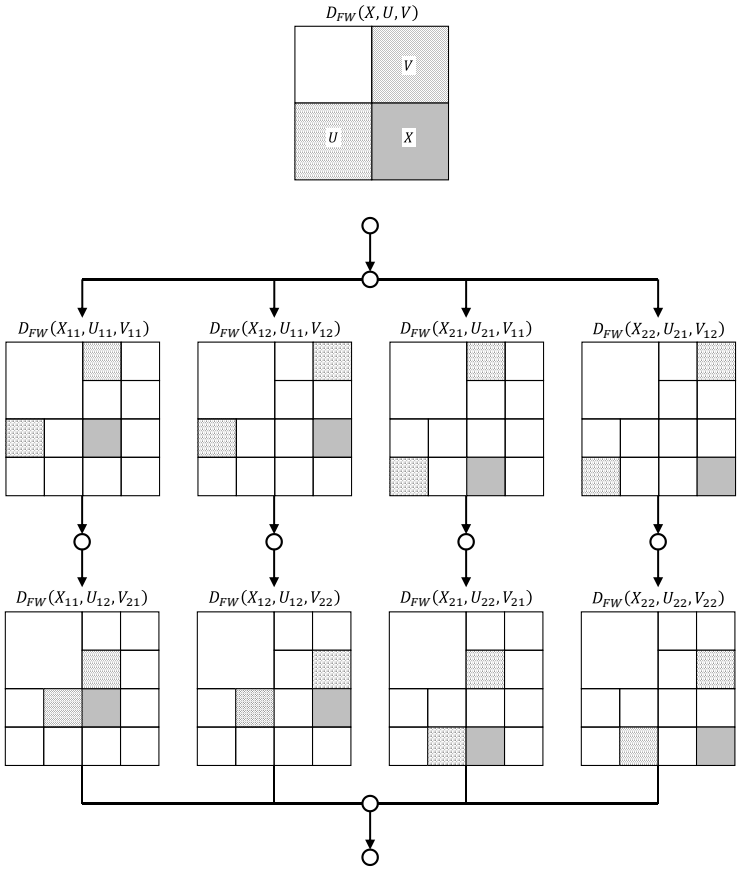

Figure 14: Function D_{FW} is invoked by functions A_{FW}, B_{FW} and C_{FW} given in Figures 11, 12 and 13, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

8 Protein accordion folding

A protein can be viewed as a string $\mathcal{P}[1:n]$ over the alphabet { *A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V* } of amino acids[1]. A protein sequence is never straight, and instead it folds itself in a way that minimizes the potential energy. Some of the amino acids (e.g., *A, I, L, F, G, P, V*)[2] are called *hydrophobic* as they do not like to be in contact with water (solvent). A desire to minimize the total hydrophobic area exposed to water is a major driving force behind the folding process. In a folded protein hydrophobic amino acids tend to clump together in order to reduce water-exposed hydrophobic area.

We assume for simplicity that a protein is folded into a 2D square lattice in such a way that the number of pairs of hydrophobic amino acids that are next to each other in the grid (vertically or horizontally) without being next to each other in the protein sequence is maximized. We also assume that the fold is always an *accordion fold* where the sequence first goes straight down, then straight up, then again straight down, and so on (see Figure 15).

[1]Amino acids: Alanine (A), Arginine (R), Asparagine (N), Aspartic acid (D), Cysteine (C), Glutamic acid (E), Glutamine (Q), Glycine (G), Histidine (H), Isoleucine (I), Leucine (L), Lysine (K), Methionine (M), Phenylalanine (F), Proline (P), Serine (S), Threonine (T), Tryptophan (W), Tyrosine (Y), Valine (V).

[2]Alanine (A), Isoleucine (I), Leucine (L), Phenylalanine (F), Glycine (G), Proline (P), Valine (V).

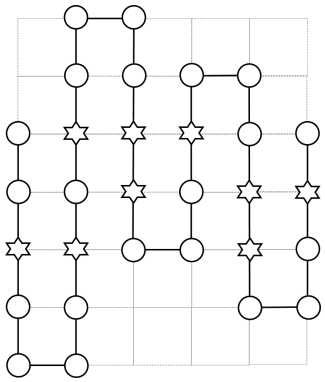

Figure 15: A protein accordion fold where each star represents a hydrophobic amino acid and each circle a hydrophilic one. The accordion score of this folded sequence is 4 which is not the maximum possible score for this sequence.

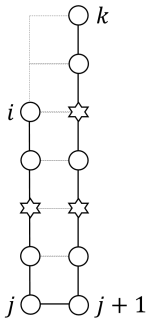

Figure 16: SCORE-ONE-FOLD(i,j,k) counts the number of aligned hydrophobic amino acids when the protein segment $\mathcal{P}[i:k]$ is folded only once at indices $(j, j+1)$. In this figure, each star represents a hydrophobic amino acid and each circle a hydrophilic one.

The recurrence below shows how to compute the optimal *accordion score* of the protein segment $\mathcal{P}[i:j]$. The optimal score for the entire sequence is given by $\max_{1<j\leq n}\{\text{SCORE}[1,j]\}$.

$$\text{SCORE}[i,j] = \begin{cases} 0 & \text{if } j \geq n-1, \\ \max_{j+1<k\leq n}\{\text{SCORE-ONE-FOLD}(i,j,k) + \text{SCORE}[j+1,k]\} & \text{otherwise.} \end{cases} \tag{20}$$

The function SCORE-ONE-FOLD(i,j,k) counts the number of aligned hydrophobic amino acids when the protein segment $\mathcal{P}[i:k]$ is folded only once at indices $(j, j+1)$. The function is illustrated graphically in Figure 16. Observe that

$$\text{SCORE-ONE-FOLD}(i,j,k) = \begin{cases} \text{SCORE-ONE-FOLD}(1,j,k) & \text{if } k \leq 2j-i+1, \\ \text{SCORE-ONE-FOLD}(1,j,2j-i+1) & \text{otherwise.} \end{cases} \tag{21}$$

Hence, in $O(n^2)$ time one can precompute an array $\text{SOF}[1:n,1:n]$ such that for all $1 \leq i < j < k-1 < n$, SCORE-ONE-FOLD$(i,j,k) = \text{SOF}[j+1, \min\{k, 2j-i+1\}]$.

Thus Recurrence 20 reduces to the following.

$$\text{SCORE}[i,j] = \begin{cases} 0 & \text{if } j \geq n-1, \\ \max_{j+1<k\leq n}\{\text{SOF}[j+1, \min\{k, 2j-i+1\}] + \text{SCORE}[j+1,k]\} & \text{otherwise.} \end{cases} \tag{22}$$

In the rest of this section we will assume for simplicity that $n = 2^t$ for some integer $t \geq 0$.

Figures 18 – 21 show a a cache-oblivious parallel algorithm for solving the protein accordion fold problem. The algorithm consists of four recursive divide-and-conquer functions named A_{fold}, B_{fold}, C_{fold} and D_{fold}. Figure 17 shows the dependency structure of the DP.

Cache Complexity on Serial Machines. For $f \in \{A,B,C,D\}$, let $Q_f(n)$ denote the cache complexity of f_{fold} on a sequence of length n. Then

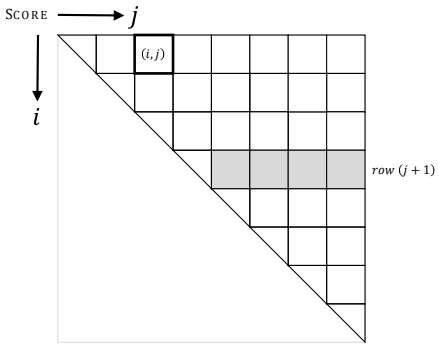

Figure 17: Dependency structure of the accordion fold DP: cell (i, j) depends on the grey cells.

$$Q_A(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_A M, \\ 2Q_A\left(\frac{n}{2}\right) + Q_B\left(\frac{n}{2}\right) + Q_C\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

$$Q_B(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_B M, \\ 4Q_B\left(\frac{n}{2}\right) + 2Q_D\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases} \qquad Q_C(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_C M, \\ 4Q_C\left(\frac{n}{2}\right) + 2Q_D\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

$$Q_D(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_D M, \\ 8Q_D\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

where, γ_A, γ_B, γ_C and γ_D are suitable constants. Solving, $Q_A(n) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$.

Span. For $f \in \{A, B, C, D\}$, let $T_f(n)$ denote the span of f_{fold} on a sequence of length n. Then

$$T_A(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_A\left(\frac{n}{2}\right) + T_B\left(\frac{n}{2}\right) + T_C\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

$$T_B(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T_B\left(\frac{n}{2}\right) + T_D\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad T_C(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2\max\left\{T_C\left(\frac{n}{2}\right), T_D\left(\frac{n}{2}\right)\right\} + \Theta(1) & \text{otherwise;} \end{cases}$$

$$T_D(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_D\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise.} \end{cases}$$

Solving, $T_A(n) = O(n \log n)$.

Cache Complexity on a Parallel Machine with Private Caches. Consider running the algorithm on p processors each with a private cache of size M and block size B. Then w.h.p. in n, the total number of cache misses incurred by the algorithm under the work-stealing scheduler is $\leq Q_A(n) + O\left(pT_A(n) \times \frac{M}{B}\right) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}} + p\left(\frac{M}{B}\right) n \log n\right)$.

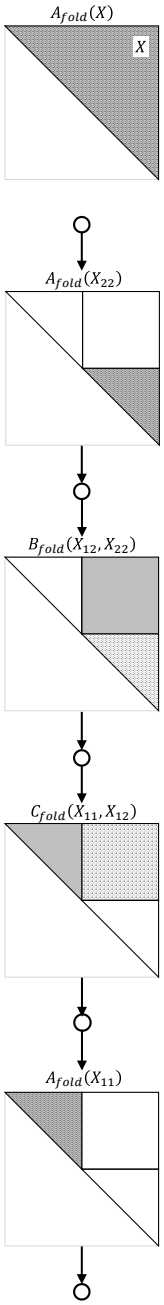

Figure 18: A cache-oblivious parallel divide-and-conquer algorithm for solving the protein accordion fold problem. Functions B_{fold} and C_{fold} are given in Figures 19 and 13, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

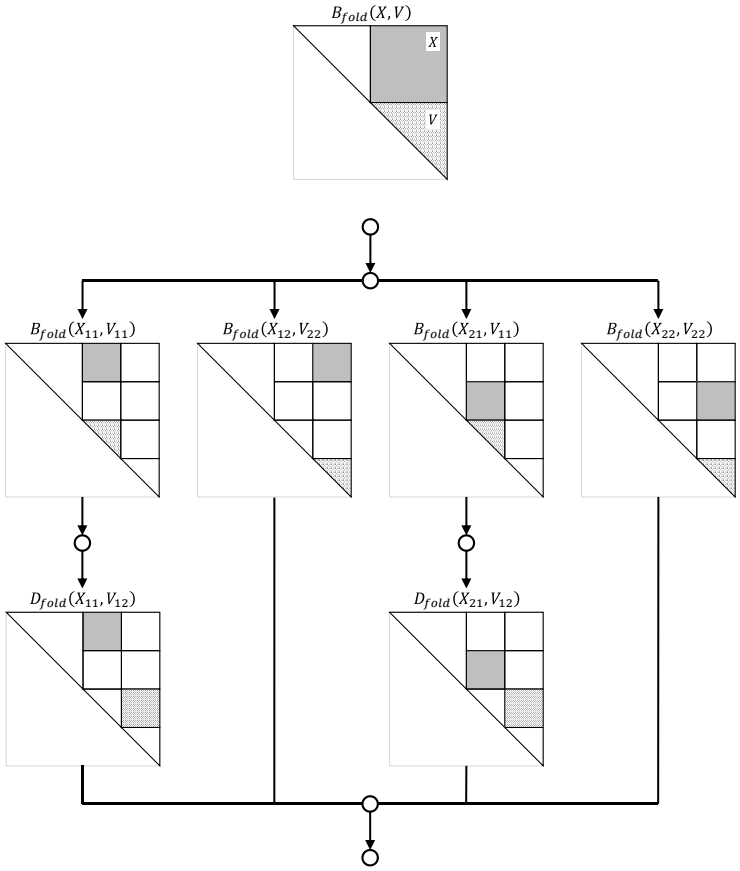

Figure 19: Function B_{fold} is invoked by function A_{fold} given in Figure 18. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

9 Basic RNA secondary structure prediction without pseudoknots

An RNA string $X = x_1x_2\ldots x_n$ forms complementary base pairs with itself, and often gives rise to interesting shapes with the following nesting condition: if $\{x_i, x_j\}$ and $\{x_k, x_l\}$ form base pairs and $i < j$, $k < l$ and $i < k$ hold then either $i < k < l < j$ or $i < j < k < l$ holds [2, 13, 15]. These shapes are known as RNA secondary structures (w/o pseudoknots). Given an RNA string, the *basic RNA secondary structure prediction* problem asks for an RNA secondary structure (w/o pseudoknots) with the maximum number of base pairs. Let $R[i, j]$ denote the maximum number of base pairs in a secondary structure formed by the RNA substrand $x_ix_{i+1}\ldots x_j$. Then the following recurrence can be used to compute $R[i, j]$ for all $i, j \in [1, n]$ [11].

$$R[i,j] = \begin{cases} 0 & \text{if } i \geq j-4, \\ \max\{\, R[i,j-1],\ P[i,j] \,\} & \text{otherwise;} \end{cases} \tag{23}$$

$$\text{where } P[i,j] \quad = \quad \max_{\substack{i<k<j-1, \\ \{x_k,x_j\} \text{ is a base pair}}} \left\{ \begin{array}{c} 1 \ + \ R[i,k-1] \\ + R[k+1,j-1] \end{array} \right\} \tag{24}$$

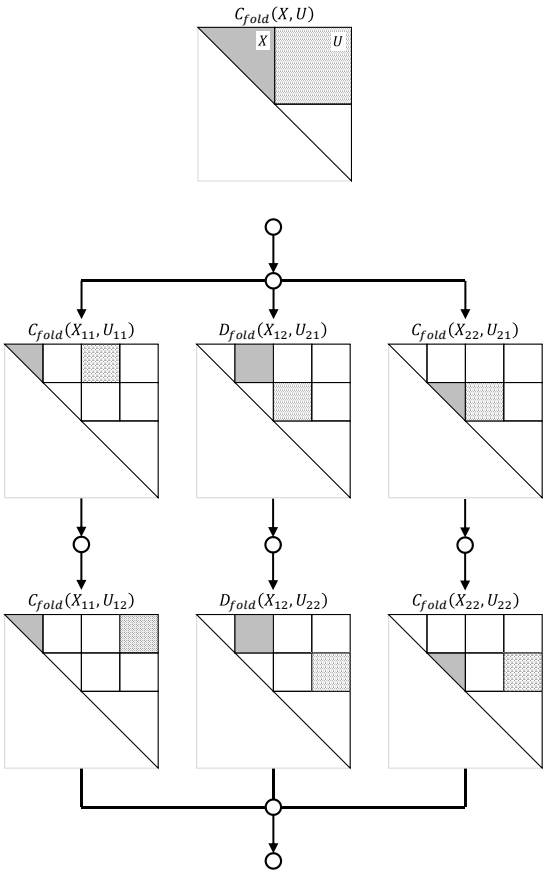

Figure 20: Function C_{fold} is invoked by function A_{fold} given in Figure 18. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

The above recurrence has the same structure as the recurrence for the *parenthesis problem* [8], and can be solved in $O(n^3)$ time and $O(n^2)$ space.

10 RNA secondary structure prediction with simple pseudoknots

A single-stranded RNA can be viewed as a string $X = x_1x_2\ldots x_n$ over the alphabet $\{A,U,G,C\}$ of bases. An RNA strand tends to give rise to interesting structures by forming *complementary base pairs* with itself. An *RNA secondary structure* (w/o pseudoknots) is a planar graph with the nesting condition: if $\{x_i,x_j\}$ and $\{x_k,x_l\}$ form base pairs and $i<j$, $k<l$ and $i<k$ hold then either $i<k<l<j$ or $i<j<k<l$ [2, 13, 15]. An *RNA secondary structure with pseudoknots* is a structure where this nesting condition is violated [2, 13].

In [2] Akutsu presented a DP to compute RNA secondary structures with maximum number of base pairs in the presence of *simple pseudoknots* (see [2] for definition) which runs in $O(n^4)$ time and $O(n^3)$ space.

We list below the DP recurrences used in Akutsu's algorithm [2]. For every pair (i_0,k_0) with $1 \leq i_0 \leq k_0-2 \leq n-2$, recurrences 25 - 29 compute the maximum number of base pairs in a pseudoknot with endpoints at the i_0-th and k_0-th residues. The value computed by recurrence 29, i.e., $S_{pseudo}(i_0,k_0)$, is the

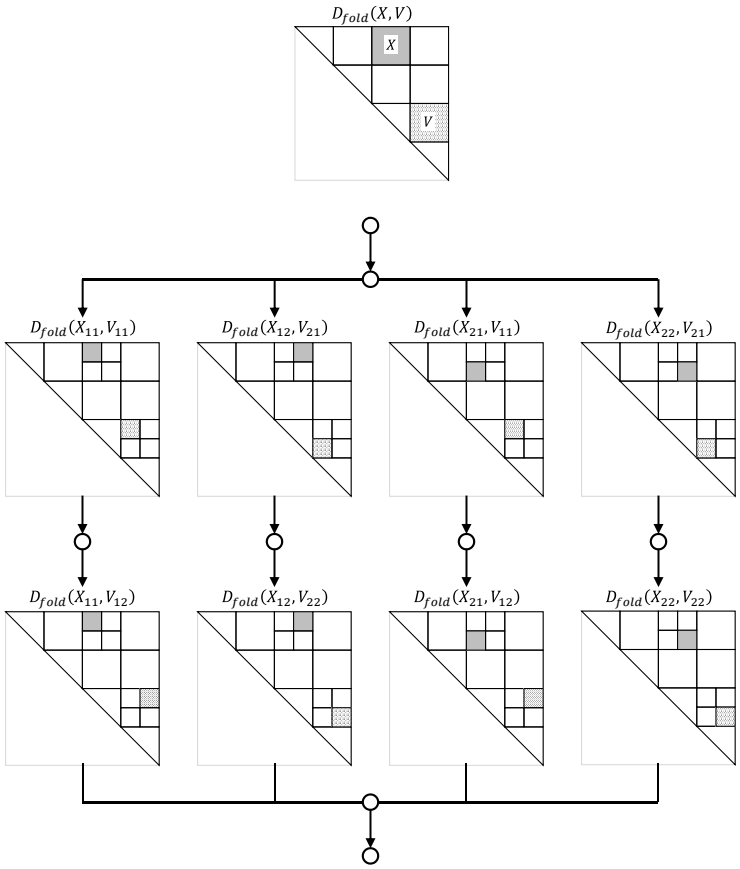

Figure 21: Function D_{fold} is invoked by functions B_{fold} and C_{fold} given in Figures 19 and 20, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

desired value. Recurrences 25 - 27 consider three locations (i,j,k) $(i_0-1 \leq i < j \leq k \leq k_0)$ on the RNA at a time. Recurrences 25 and 26 correspond to cases where $\{x_i,x_j\}$ and $\{x_j,x_k\}$ form base pairs, respectively, while recurrence 27 handles the case where neither $\{x_i,x_j\}$ nor $\{x_j,x_k\}$ forms a base pair. The variable $S_{MAX}(i,j,k)$ in recurrence 28 contains the maximum score for the triple (i,j,k). In recurrences 25 and 26, $v(x_s,y_t)=1$ if $\{x_s,y_t\}$ form a base pair, otherwise $v(x_s,y_t)=-\infty$. All uninitialized entries are assumed to have value 0.

$$S_L(i,j,k) = \begin{cases} v(x_i,x_j) & \text{if } i_0 \leq i < j \geq k, \\ \quad v(a_i,a_j) & \\ +S_{MAX}(i-1,j+1,k) & \text{if } i_0 \leq i < j < k. \end{cases} \tag{25}$$

$$S_R(i,j,k) = \begin{cases} v(x_j,x_k) & \text{if } i_0-1=i<j-1=k-2, \\ \quad v(a_j,a_k) & \\ +S_{MAX}(i,j+1,k-1) & \text{if } i_0 \leq i < j < k. \end{cases} \tag{26}$$

$$S_M(i,j,k) = \max \left\{ \begin{array}{c} S_L(i-1,j,k), \\ S_M(i-1,j,k), \\ S_{MAX}(i,j+1,k), \\ S_M(i,j,k-1), \\ S_R(i,j,k-1) \end{array} \right\} \text{ if } i_0 \leq i < j < k. \tag{27}$$

$$S_{MAX}(i,j,k) = \max \{ S_L(i,j,k),\ S_M(i,j,k),\ S_R(i,j,k) \} \tag{28}$$

$$S_{pseudo}[i_0,k_0] = \max_{i_0 \leq i < j < k \leq k_0} \{ S_{MAX}(i,j,k) \} \tag{29}$$

After computing all entries of S_{MAX} for a fixed i_0, all $S_{pseudo}[i_0,k_0]$ values for $k_0 \geq i_0+2$ can be computed using equation 29 in $O(n^3)$ time and space Since there are $n-2$ possible values for i_0, all $S_{pseudo}[i_0,k_0]$ can be computed in $O(n^4)$ time and $O(n^3)$ space.

Finally, the following recurrence computes the optimal score $S[1,n]$ for the entire structure.

$$S[i,j] = \max \left\{ \begin{array}{c} S_{pseudo}[i,j],\ S[i+1,j-1]+v(a_i,a_j), \\ \max_{i<k\leq j} \{S[i,k-1],S[k,j]\} \end{array} \right\} \tag{30}$$

Iterative evaluation of recurrence 30 requires $O(n^3)$ time and $O(n^2)$ space.

Space Reduction. We now describe a space reduction result given in [4]. A similar method was suggested in [12]. Observe that evaluating recurrence 29 requires retaining all $O(n^3)$ values computed by recurrence 28. We avoid using this extra space by computing all required $S_{pseudo}[i_0,k_0]$ values on the fly while evaluating recurrence 28. We achieve this by introducing recurrence 31, replacing recurrence 29 with recurrence 32 for S'_{pseudo}, and using S'_{pseudo} instead of S_{pseudo} for evaluating recurrence 30. Intuitively, the variable $S_P(i,j,k)$ in recurrence 31 stores the maximum score among all triples (i,j',k) with $j' \geq j$. All uninitialized entries in recurrences 31 and 32 are assumed to have value $-\infty$.

$$S_P(i,j,k) = \left\{ \begin{array}{ll} \max \left\{ \begin{array}{c} S_{MAX}(i,j,k), \\ S_P(i,j+1,k) \end{array} \right\} & \text{if } i_0 \leq i < j < k, \\ S_P(i,j+1,k) & \text{if } i_0 \leq i \geq j < k. \end{array} \right. \tag{31}$$

$$S'_{pseudo}[i_0,k_0] = \max \left\{ \begin{array}{c} S'_{pseudo}[i_0,k_0-1], \\ \max\limits_{i_0 \leq i < k_0-1} \left\{ S_P(i,i_0+1,k_0) \right\} \end{array} \right\} \text{ if } k_0 \geq i_0+2. \tag{32}$$

The proof that recurrence 32 computes exactly the same values as recurrence 29 appears in [4].

Now observe that in order to evaluate recurrence 32 we only need the values $S_P(i,j,k)$ for $j = i_0+1$, and each entry (i,j,k) in recurrences 25 - 28 and 31 depends only on entries $(\cdot,j,\cdot)$ and $(\cdot,j+1,\cdot)$. Therefore, we will evaluate the recurrences for $j=n$ first, then for $j=n-1$, and continue downto $j=i_0+1$. Observe that in order to evaluate for $j=j'$ we only need to retain the $O(n^2)$ entries computed for $j=j'+1$. Thus for a fixed i_0 all $S_P(i,i_0+1,k)$ and consequently all relevant $S'_{pseudo}[i_0,k_0]$ can be computed using only $O(n^2)$ space, and the same space can be reused for all n values of i_0.

11 Cocke-Younger-Kasami (CYK) algorithm

We are given a string $X = \langle x_1, x_2, \ldots, x_n \rangle$ and a context-free grammar $G = (V, \sum, R, S)$, where V is a set of variables (or non-terminal symbols), $\sum$ is a finite set of terminal symbols, $R = \{R_1, R_2, \ldots, R_t\} : V \rightarrow (V \cup \sum)^*$ is a finite set of rules, and S is a start variable chosen from V. We set $P[i, j, c]$ to ***true*** provided substring $X_{ij} = x_i x_{i+1} \ldots x_{i+j-1}$ can be generated from rule $R_c \in R$, and to ***false*** otherwise. Then

$$P[i,j,c] = \begin{cases} \textbf{\textit{true}} & \text{if } j = 1 \text{ and } R_c \rightarrow x_i, \\ \textbf{\textit{true}} & \text{if } P[i,k,a] = P[i+k, j-k, b] = \textbf{\textit{true}}, k \in [1, i-1] \text{ and } R_c \rightarrow R_a R_b, \\ \textbf{\textit{false}} & \text{otherwise.} \end{cases} \tag{33}$$

In Figures 23, 24 and 25, a cache oblivious parallel algorithm is given for solving the CYK problem. The algorithm consists of three recursive divide-and-conquer functions named A_{cyk}, B_{cyk} and C_{cyk}. Figure 22 shows the dependency structure of the DP.

Cache Complexity on Serial Machines. For $f \in \{A, B, C\}$, let $Q_f(n)$ denote the cache complexity of f_{cyk} on a matrix of size $n \times n$. Then

$$Q_A(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_A M, \\ 2Q_A\left(\frac{n}{2}\right) + Q_B\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

$$Q_B(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_B M, \\ 4\left(Q_B\left(\frac{n}{2}\right) + Q_C\left(\frac{n}{2}\right)\right) & \text{otherwise;} \end{cases} \qquad Q_C(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma_C M, \\ 8Q_C\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

where, γ_A, γ_B and γ_C are suitable constants. Solving, $Q_A(n) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$.

Span. For $f \in \{A, B, C\}$, let $T_f(n)$ denote the span of f_{cyk} on a matrix of size $n \times n$. Then

$$T_A(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T_A\left(\frac{n}{2}\right) + T_B\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

$$T_B(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T_B\left(\frac{n}{2}\right) + 3T_C\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad T_C(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_C\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

Solving, $T_A(n) = O(n)$.

Cache Complexity on a Parallel Machine with Private Caches. Consider running the algorithm on p processors each with a private cache of size M and block size B. Then w.h.p. in n, the total number of cache misses incurred by the algorithm under the work-stealing scheduler is $\leq Q_A(n) + O\left(pT_A(n) \times \frac{M}{B}\right) = O\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}} + p\left(\frac{M}{B}\right)n\right)$.

12 Knapsack Problem

We are given a knapsack of capacity W, a set of n items $\{x_1, x_2, \ldots, x_n\}$, where item x_i has a value v_i and weight w_i. Let $K[i, j]$ denote the maximum value of a subset of $\{x_1, x_2, \ldots, x_i\}$ that is less than or equal to j.

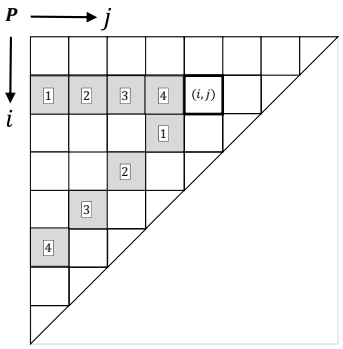

Figure 22: Dependency structure of the CYK DP: cell (i, j) depends on the grey cells. Cell numbered k on the horizontal grey line is paired with cell numbered k on the diagonal grey line during the update of cell (i, j).

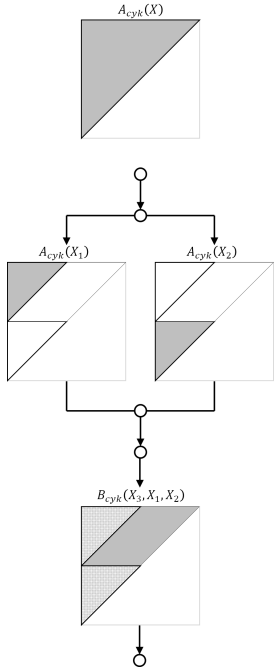

Figure 23: A cache-oblivious parallel divide-and-conquer algorithm for solving the CYK problem. The initial call to the function is $A_{cyk}(X)$. Functions B_{cyk} and C_{cyk} are given in Figures 24 and 25, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

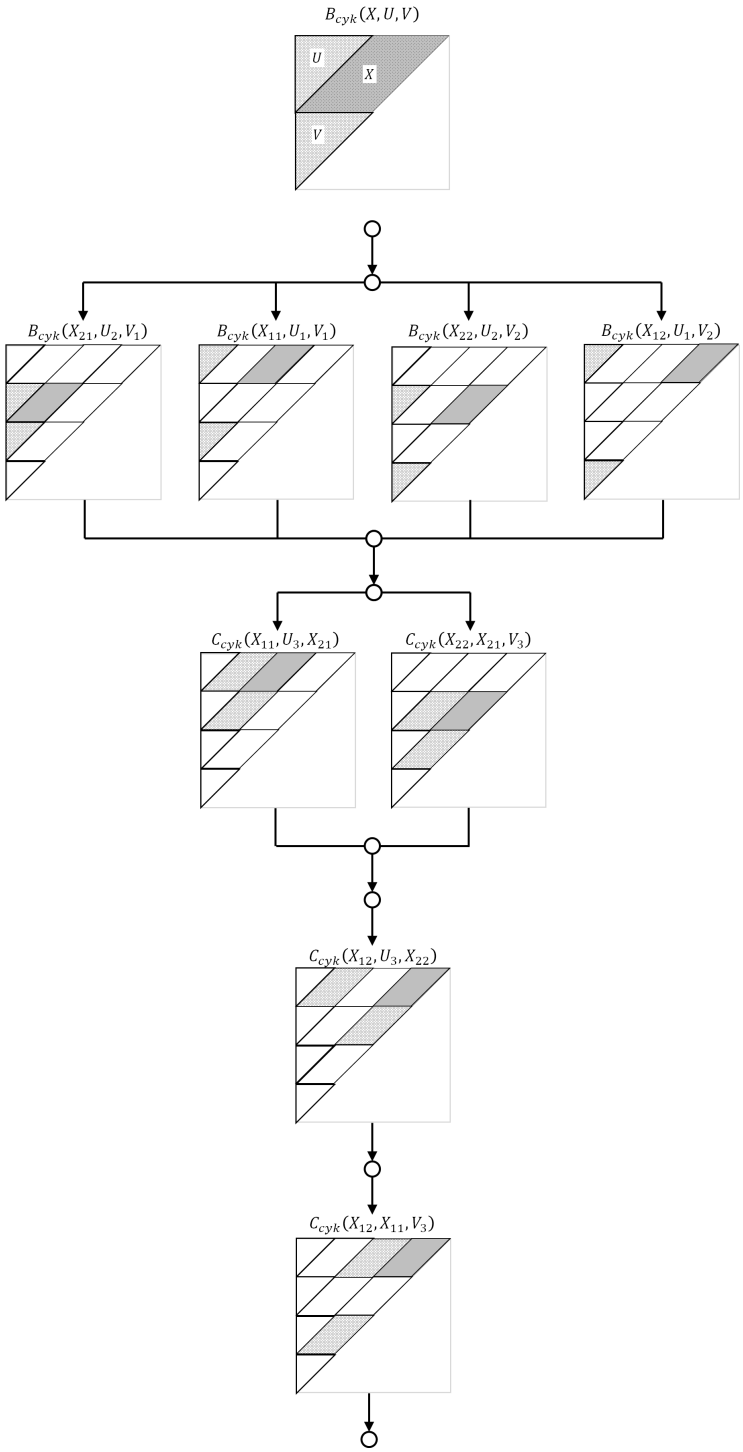

Figure 24: Function B_{cyk} is invoked by function A_{cyk} given in Figure 23 which is a cache-oblivious parallel divide-and-conquer algorithm for solving the CYK problem. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

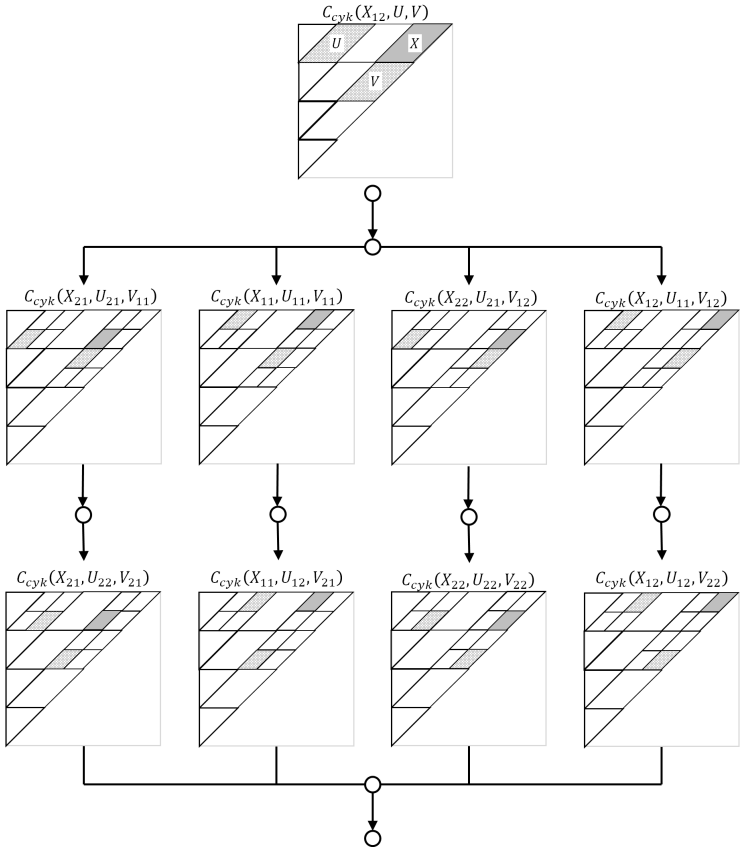

Figure 25: Function C_{cyk} is invoked by functions A_{cyk} and B_{cyk} given in Figures 23 and 24, respectively. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

Then

$$K[i,j] = \begin{cases} 0 & \text{if } j = 0, \\ K[i-1,j] & \text{if } w_i > w, \\ \max(K[i-1,j], K[i-1,j-w_i] + v_i) & \text{otherwise.} \end{cases} \tag{34}$$

Subset-sum Problem. We are given a multiset $S = \{x_1, x_2, \ldots, x_n\}$ of n integers. We set $P[i,j]$ to ***true*** if a subset of $\{x_1, x_2, \ldots, x_i\}$ sums to j, and to ***false*** otherwise. Then

$$P[i,j] = \begin{cases} \textbf{\textit{true}} & \text{if } j = 0, \\ \textbf{\textit{true}} & \text{if } P[i-1,j] = \textbf{\textit{true}} \text{ or } P[i-1,j-x_i] = \textbf{\textit{true}}, \\ \textbf{\textit{false}} & \text{otherwise.} \end{cases} \tag{35}$$

13 Viterbi Algorithm

We are given an observation space $O = \{o_1, o_2, \ldots, o_N\}$, state space $S = \{s_1, s_2, \ldots, s_K\}$, observations $Y = \{y_1, y_2, \ldots, y_T\}$, transition matrix A, where $A[i,j]$ is the transition probability of transiting from s_i to s_j,

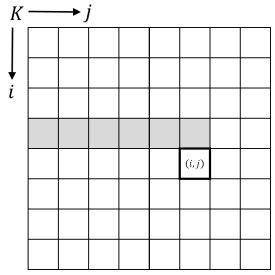

Figure 26: Dependency structure of the Knapsack DP: cell (i, j) depends on the grey cells.

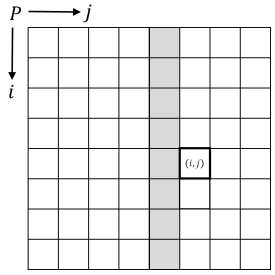

Figure 27: Dependency structure of the Viterbi DP: cell (i, j) depends on the grey cells.

emission matrix B, where $B[i, j]$ is the probability of observing o_j from s_i, initial probability array I, where $I[i]$ is the probability that $x_i = s_i$. Then,

$$P[i, j] = \begin{cases} I[i] \times B[i, y_1] & \text{if } j = 1, \\ \max_{k \in [1,K]}(P[k, j-1] \cdot A[k, i] \cdot B[i, y_j]) & \text{otherwise.} \end{cases} \tag{36}$$

14 Eggs Problem

A building has n floors and we are given k eggs. There might be a threshold floor in the building from and below which when an egg is dropped, the egg does not break, and above which when the egg is dropped, the egg breaks. Let $D[i][j]$ denote the minimum number of drops for first i floors and using j eggs. Then,

$$D[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \min_{k \in [1,i]}(\max(D[k-1, j-1], D[i-k, j])) & \text{otherwise.} \end{cases} \tag{37}$$

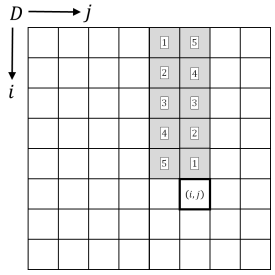

Figure 28: Dependency structure of the Eggs DP: cell (i, j) depends on the grey cells.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45–62, 2000.

[3] C. Cherng and R. E. Ladner. Cache efficient simple dynamic programming. In *Proc. of the Int. Conference on the Analysis of Algorithms*, pages 49–58, 2005.

[4] Rezaul Chowdhury, Hai-Son Le, and Vijaya Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(3):495–510, July-September 2010.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[6] R. W. Floyd. Algorithm 97 (SHORTEST PATH). *Commun. ACM*, 5(6):345, 1962.

[7] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[8] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $o(1)$ dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[9] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[10] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM J. Comput.*, 16(4):628–638, 1987.

[11] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesely, 2005.

[12] R.B. Lyngsϕ and C.N. Pedersen. RNA pseudoknot prediction in energy-based models. *J. Comput. Biol.*, 7(3-4):409–427, 2000.

[13] E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *J. Mol. Biol.*, 285(5):2053–2068, 1999.

[14] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[15] M. S. Waterman. *Introduction to Computational Biology*. Chapman & Hall, London, UK, 1995.