DP problems of interest

Manohar Jonnalagedda, Thierry Coppey

1 Introduction

1.1 Definitions

- Block of computation: a block is simply a part of the DP matrix that we want to compute.
- Wavefront: this is the place around which computation happen, typically. There should be some memory to store intermediate information between block of computations

1.2 Problems classification

In the literature dynamic programming problems (DP) are classified according to two criteria:

Monadic / polyadic

- Monadic: on the right hand-side of the recurrence formula, only one term appears. For instance, Smith-Waterman with constant penalty is monadic

$$M_{(i,j)} = \max \begin{cases} 0 \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) \\ M_{(i-1,j)} - d \\ M_{(i,j-1)} - d \end{cases}$$

- **Polyadic:** when multiple terms of the recurrence occur in the right and-side of the recurrence formula. For instance Fibonacci is polyadic:

$$F(n) = F(n-1) + F(n-2)$$

Serial/non-serial

- **Serial:** when the solution depends only of a fixed number of immediately previous solutions (i.e. neighbor cells). For instance Fibonacci is serial.
- **Non-serial:** when the solution of a level depends of an arbitrary number of previous solutions. Typically Smith-Waterman with arbitrary gap penalty and Nussinov are non-serial:

$$M_{(i,j)} = \max \begin{cases} \dots \\ M_{(i,j-1)} \\ \max\limits_{i<k<j} [M_{(i,k)} + M_{(k+1,j)}] \end{cases}$$

1.3 Simplifications

1.3.1 Calculus

In some special case, it is possible to transform a non-serial problem into a serial problem, if we can embed the non-serial term into an additional aggregation matrix. For example, we can

transform

$$M_{(i,j)} = \max \begin{cases} \max\limits_{k<i} M_{(k,j)} \\ \sum\limits_{k<i,l<j} M_{(k,l)} \end{cases} \implies M_{(i,j)} = \max \begin{cases} C_{(k,j)} \\ A_{(i-1,j-1)} \end{cases}$$

Where C is a matrix that stores the maximum along the column and A is a matrix that stores the sum of the array of the previous elements. They can be easily computed with the additional recurrence.

$$\begin{aligned} C_{(i,j)} &= \max(C_{(i-1,j)}, M_{(i,j)}) \\ A_{(i,j)} &= A_{(i-1,j)} + A_{(i,j-1)} - A_{(i-1,j-1)} + M_{(i,j)} \end{aligned}$$

This simplification avoids the non-serial dependencies at the cost of an additional storage at the wavefront.

1.3.2 Precomputations

When a calculus computation is impossible, it might be worth to interleave a computation phase that will aggregate some of the results that are necessary to the computation block. For instance, for Nussinov's $\max_{i<k<j}[M_{(i,k)} + M_{(k+1,j)}]$ we can precompute it for all rows and columns of the block, and for all elements that are not part of the block, and pass these partial results together at the block launch.

On GPU, this could be done by interleaving a new kernel for this specific purpose, on FPGA, this could be done by preparing the maximums in another memory area whose pointer will later be passed to the co-processor.

2 Problems of interest

We describe problems structures: inputs, cost matrices, backtracking matrix. They all have an alphabet (that hopefully corresponds to a maximal bit-size) and matrixes dimensions are specified by number of indices and their number of elements is usually the same as the input. Unless otherwise specified, number are all unsigned integers. We describe the problem processing in terms of both initialization and recurrences.

2.1 Smith-Waterman (simple)

Problem: matching two strings S, T.

Alphabets:

- Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
- Cost matrix: $\Sigma(M) = \text{integers}\{0..n\}, n = \max(\text{cost}) \cdot \min(\text{length}(S), \text{length}(T))$
- Backtrack matrix: $\Sigma(B) = \{stop, W, N, NW\}$

Initialization:

- Cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
- Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = stop$.

Recurrence:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} 0 & stop \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) & NW \\ M_{(i-1,j)} - d & N \\ M_{(i,j-1)} - d & W \end{array} \right\} = B_{(i,j)}$$

2.2 Smith-Waterman (with gap extension at different cost)

Problem: matching two strings S, T.

Alphabets:

- Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
- Cost matrix: $\Sigma(M) = \text{integers}\{0..n\}, n = \max(\text{cost}) \cdot \min(\text{length}(S), \text{length}(T))$
- Backtrack matrix: $\Sigma(B) = \{stop, W, N, NW\}$

Initialization:

- Cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
- Gap opening matrix: $E_{(i,0)} = 0, 0 \leq i \leq \text{length}(S)$
- Gap extending matrix: $E_{(0,j)} = 0, 0 \leq j \leq \text{length}(T)$
- Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = stop$.

Recurrence for the cost matrix:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} 0 & stop \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) & NW \\ E_{(i,j)} & N \\ F_{(i,j)} & W \end{array} \right\} = B_{(i,j)}$$

Recurrence for the gap opening/extending matrices:

$$E_{(i,j)} = \max \left\{ \begin{array}{l|l} M_{(i,j-1)} - \alpha & NW \\ E_{(i,j-1)} - \beta & N \end{array} \right\} = B_{(i,j)}$$

$$F_{(i,j)} = \max \left\{ \begin{array}{l|l} M_{(i-1,j)} - \alpha & NW \\ E_{(i-1,j)} - \beta & N \end{array} \right\} = B_{(i,j)}$$

Otherwise written as:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} 0 & stop \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) & NW \\ \max_{1 \leq k \leq j-1} M_{(i,k)} - \alpha - (j-1-k) \cdot \beta & N \\ \max_{1 \leq k \leq i-1} M_{(k,j)} - \alpha - (i-1-k) \cdot \beta & W \end{array} \right\} = B_{(i,j)}$$

2.3 Convex polygon triangulation

Problem: triangulating a polygon with least cost overall for added edges

Alphabets:

- Input: $\Sigma(S)$ = a matrix of costs for every possible edge in the polygon. The size of the input is $\frac{n^2}{2}$, where n is the number of vertices in the polygon.

- Cost matrix: $\Sigma(M) = \mathbb{N}$. The sum of cost of edges is not "bounded".
- Backtrack matrix: $\Sigma(B) = \{n\}$

Recurrence:

$$M_{(i,j)} = \max_{i<k<j} M_{(i,k)} + M_{(k+1,j)} + c(i,k)|k = B_{(i,j)}$$

It is interesting to note that even in the sequential world, this problem is best solved by filling the diagonals, ie. computing sub-solutions for all polygons on size k before those of size $k+1$.

2.4 Matrix chain multiplication

Problem: find an optimal parenthesizing of the multiplication of n matrices A_i of dimension $r_i \times c_i$.

Alphabets:

- Input: matrix size is defined as pairs of integers (r_i, c_i).
- Cost matrix: the cost is an integer (might blow up, use a float or a double?).
- Backtrack matrix: $\Sigma(B) = \{stop\} \cup \{0..n\}$ with n the input length.

Initialization (line=length of sequence, column=start):

- Cost matrix: $M_{(0,j)} = 0$.
- Backtrack matrix: $B_{(0,j)} = stop$.

Recurrence for the cost matrix:

$$M_{(i,j)} = \min_{i \leq k < j} \left\{ \; M_{(i,k)} + M_{(k,j)} + r_i \cdot r_k \cdot col_j \;\middle|\; k \; \right\} = B_{(i,j)}$$

2.5 Nussinov algorithm

Problem: folding a RNA string S over itself.

Alphabets:

- Input: $\Sigma(S) = \{A, C, G, U\}$.
- Cost matrix: $\Sigma(M) = \{0..n\}, n = \text{length}(S)/2$
- Backtrack matrix: $\Sigma(B) = \{stop, W, S, SW, 1..n\}$

Initialization:

- Cost matrix: $\left\{ \begin{array}{l} M_{(i,i)} = 0 \\ M_{(i,i-1)} = 0 \end{array} \right. \forall i \in 1..\text{length}(S)$
- Backtrack matrix: $\left\{ \begin{array}{l} B_{(i,i)} = stop \\ B_{(i,i-1)} = stop \end{array} \right. \forall i \in 1..\text{length}(S)$

Recurrence:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} M_{(i+1,j-1)} + \omega(i,j) & SW \\ M_{(i+1,j)} & S \\ M_{(i,j-1)} & W \\ \max_{i<k<j} M_{(i,k)} + M_{(k+1,j)} & k \end{array} \right\} = B_{(i,j)}$$

With $\omega(i,j) = 1$ if i, j are complementary. 0 otherwise.

«««< HEAD

2.6 Zuker folding

XXX: we let this aside for the moment as the recurrence has too many unknown parameters
Problem: folding a RNA string S over itself.

Alphabets:

- Input: $\Sigma(S) = \{A, C, G, U\}$.
- Cost matrix: $\Sigma(W) = \Sigma(V) = \Sigma(F) = \{0..n\}, n =$
- Backtrack matrix: $\Sigma(B) = \{\}$

Initialization:

- Cost matrices:
  - xxx
- Backtrack matrix: B

Recurrence:

$$W_{(i,j)} = \min \begin{cases} W_{(i+1,j)} + b \\ W_{(i,j-1)} + b \\ V_{(i,j)} + \delta(S_i, S_j) \\ \min_{i<k<j} W_{(i,k)} + W_{(k+1,j)} \end{cases}$$

$$V_{(i,j)} = \min \begin{cases} \infty & \text{if}(S_i, S_j) \text{ is not a base pair} \\ eh(i,j) + b & \text{otherwise} \\ V_{(i+1,j-1)} + es(i,j) & \\ VBI(i,j) & \\ \min_{i<k<j-1}\{W_{(i+1,k)} + W_{(k+1,j-1)}\} + c & \end{cases}$$

$$VBI(i,j) = \min_{i<i'<j'<j}\{V_{(i',j')} + ebi(i,j,i',j')\} + c$$

$$F_{(j)} = \min \begin{cases} F_{(j-i)} \\ \min_{1 \leq i < j}(V_{(i,j)} + F_{(j-1)}) \end{cases} \text{(Free Energy)}$$

2.7 Comments

VBI stores scores for internal loops and bulges. Notice in VBI, two moving indices i', j'. In general this gives a quadratic complexity for that particular recurrence, making the overall complexity of the algorithm $O(n^4)$. In practice (as read in Arpith's work), we have that $i' - i + j' - j < 30$, so the complexity is reduced to $O(n^3)$, albeit with a large constant.

2.8 Zuker folding

XXX: we let this aside for the moment as the recurrence has too many unknown parameters
Problem: folding a RNA string S over itself.

Alphabets:

- Input: $\Sigma(S) = \{A, C, G, U\}$.
- Cost matrix: $\Sigma(W) = \Sigma(V) = \Sigma(F) = \{0..n\}, n =$
- Backtrack matrix: $\Sigma(B) = \{\}$

Initialization:

- Cost matrices:
  - xxx
- Backtrack matrix: B

Recurrence:

$$W_{(i,j)} = \min \begin{cases} W_{(i+1,j)} + b \\ W_{(i,j-1)} + b \\ V_{(i,j)} + \delta(S_i, S_j) \\ \min_{i<k<j} W_{(i,k)} + W_{(k+1,j)} \end{cases}$$

$$V_{(i,j)} = \min \begin{cases} \infty & \text{if}(S_i, S_j) \text{ is not a base pair} \\ eh(i,j) + b & \text{otherwise} \\ V_{(i+1,j-1)} + es(i,j) & \\ VBI(i,j) & \\ \min_{i<k<j-1}\{W_{(i+,k)} + W_{(k+1,j-1)}\} + c & \end{cases}$$

$$F_{(j)} = \min \begin{cases} F_{(j-i)} \\ \min_{1 \le i < j}(V_{(i,j)} + F_{(j-1)}) \end{cases} \text{(Free Energy)}$$

$$VBI(i,j) = \min_{i<i'<j'<j}\{V_{(i',j')} + ebi(i,j,i',j')\} + c$$