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 (it accesses only 2 cells backward).
- **Non-serial:** when the solution depends of an arbitrary number of previous solutions. Typically Smith-Waterman with arbitrary gap penalty and Nussinov are non-serial:

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

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:

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

Where the matrix C stores the maximum along the column and matrix A stores the sum of the array of the previous elements. Both can be easily computed with an 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 extra storage in the wavefront, unfortunately, it might be applicable only for some special cases.

1.3.2 Precomputations

When a calculus transformation 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 term $\max_{i<k<j}[M_{(i,k)} + M_{(k+1,j)}]$, we can precompute it over all rows and columns of the block, and for all elements that are not part of the block, then pass these partial results together at the block launch to finish the computation.

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. We may notice that since this phase is at the same time necessary for both architecture, and can be run independently, we can both execute them concurrently and mix between architectures (use CUDA for pre-computation and FPGA for actual tile computation for instance).

2 Problems of interest

We describe problems structures: inputs, cost matrices and backtracking matrix. These all have an alphabet (that must be bounded in terms of bit-size). Unless otherwise specified, we adopt the following conventions:

- Matrices dimensions are implicitly specified by number of indices and their number of elements is usually the same as the input length.
- Number are all unsigned integers
- Problem dimension is m, n (or n) indices i, j ranges are respectively $0 \leq i < m$, $0 \leq j < n$.
- Unless otherwise specified, the recurrence applies to all non-initialized matrix elements.

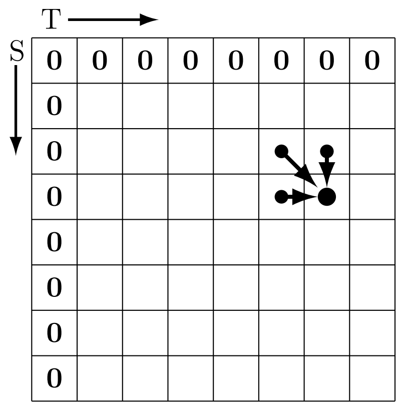
We describe the problem processing in terms of both initialization and recurrences.

2.1 Smith-Waterman (simple)

1. Problem: matching two strings S, T with $|S| = m, |T| = n$.
2. Matrices: $M_{m\times n}, B_{m\times n}$
3. Alphabets:
   - Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
   - Cost matrix: $\Sigma(M) = [0..z], z = \max(\mathrm{cost}(\_)) \cdot \min(m, n)$
   - Backtrack matrix: $\Sigma(B) = \{\mathit{stop}, W, N, NW\}$
4. Initialization:
   - Cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
   - Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = \mathit{stop}$.
5. Recurrence:

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

6. Backtracking: starts from the cell $\max\{M_{(m,j)} \cup M_{(i,n)}\}$
7. Visualisation: by convention, we put the longest string vertically ($m \geq n$):



8. Optimizations:
   - In serial (monadic) problems we can avoid building the matrix M by only maintaining the 3 last diagonals in memory (one for the diagonal element, one for horizontal/vertical, and one being built). This construction extends easily to polyadic problems where we need to maintain $k + 2$ diagonals in memory where k is the maximum backward lookup.

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

1. Problem: matching two strings S, T with $|S| = m, |T| = n$.
2. Matrices: $M_{m\times n}, E_{m\times n}, F_{m\times n}, B_{m\times n}$
3. Alphabets:
   - Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
   - Cost matrices: $\Sigma(M) = \Sigma(E) = \Sigma(F) = [0..z], z = \max(\text{cost}(\_)) \cdot \min(m, n)$
   - Backtrack matrix: $\Sigma(B) = \{\mathit{stop}, W, N, NW\}$
4. Initialization:
   - No gap cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
   - T-gap extension cost matrix: $E_{(i,0)} = 0$ *«eat S chars only»*
   - S-gap extension cost matrix: $F_{(0,j)} = 0$
   - Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = \mathit{stop}$.
5. Recurrence for the cost matrices:

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

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

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

That can be written alternatively as:

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

Although the latter notation seems more explicit, it introduces non-serial dependencies that the former set of recurrences is free of. So we need to implement the former rules whose kernel is

$$[M; E; F]_{(i,j)} = f_{\text{kernel}}([M; E]_{(i,j-1)}, [M; F]_{(i-1,j)}, M_{(i-1,j-1)})$$

Notice that this recurrence is very similar to Smith-Waterman (simple) except that we propagate 3 values (M, E, F) instead of a single one (M).

6. Backtracking: same as Smith-Waterman (simple)
7. Visualisation: same as Smith-Waterman (simple)
8. Optimizations: same as Smith-Waterman (simple)

2.3 Smith-Waterman with arbitrary gap cost

1. Problem: matching two strings S, T with $|S| = m, |T| = n$ with an arbitrary gap function $g(x) \geq 0$ where x is the size of the gap. Without loss of generality, let $m \geq n$[1]. Example penalty function could be[2] $g(x) = m - x$.
2. Matrices: $M_{m\times n}, B_{m\times n\times m}$
3. Alphabets:
   - Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
   - Cost matrix: $\Sigma(M) = [0..z], z = \max(\mathrm{cost}(\_)) \cdot \min(m, n)$
   - Backtrack matrix: $\Sigma(B) = \{stop, NW, N_{\{0..m\}}, W_{\{0..n\}}\}$
4. Initialization:
   - No gap cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
   - Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = stop$.
5. Initialization:
6. Recurrence:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} 0 & stop \\ M_{(i-1,j-1)} + \mathrm{cost}(S(i), T(j)) & NW \\ \max_{1\leq k\leq j-1} M_{(i,j-k)} - g(k) & N_k \\ \max_{1\leq k\leq i-1} M_{(i-k,j)} - g(k) & W_k \end{array} \right\} = B_{(i,j)}$$

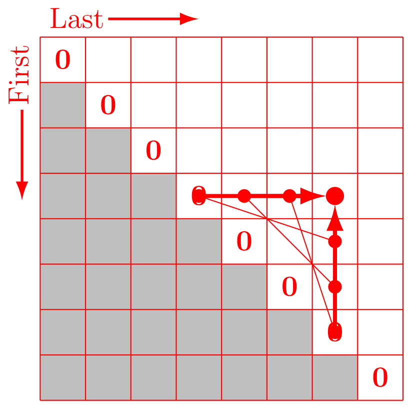
7. Backtracking:
8. Visualisation:
9. Optimizations:

[1]Otherwise if $|T| > |N|$ we only need to swap both the inputs and backtracking pairs.

[2]Intuition: long gaps penalize less, at some point, one large gap is better than matching and smaller gaps.

2.4 Convex polygon triangulation

1. Problem: triangulating a polygon of n vertices with least total cost for added edges. We denote the cost of adding an edge between the pair of edges i, j by $S(i, j)$, Where $S_{n\times n}$ is a lower triangular matrix compacted in memory (rows are contiguous) with a 0 diagonal that is omitted [3], hence $|S| = \frac{n^2}{2} = N$.
2. Matrices: $M_{n\times n}, B_{n\times n}$ *«first edge, last edge»* upper triangular including main diagonal
3. Alphabets:
   - Input: $\Sigma(S_{(i,j)}) = \{0..m\}$ with $m = \max_S(i, j)\forall i, j$ determined at runtime[4].
   - Cost matrix: $\Sigma(M) = \{0..z\}$ with $z = m \cdot (n - 2)$ (we add at most $n - 2$ edges).
   - Backtrack matrix: $\Sigma(B) = \{stop, 0..n\}$ (the index of the edge we add)
4. Initialization: $M_{(i,i)} = 0, B_{(i,i)} = stop \quad \forall i$
5. Recurrence:
$$M_{(i,j)} = \left\{ \max_{i<k<j} M_{(i,k)} + M_{(k+1,j)} + S(i, k) \;\middle|\; k \right\} = B_{(i,j)}$$
   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 $k + 1$.
6. Backtracking: Starting at $B_{(0,n)}$ and let current point you backtrack drawing an edge from the current position to the backtracking position ??
7. Visualisation: in its traditional form the matrix is upper triangular with diagonal excluded.



8. Optimizations: XXX we need to rotate that matrix to progress in the same direction as usual, that is towards bottom right.

[3] Arbitrary convention for both architectural implementation and code generator. Rationale: in lower triangular matrix, element address is independent of the matrix size.

[4] We need to scan/have stats about S and that's where LMS plays a role

2.5 Matrix chain multiplication

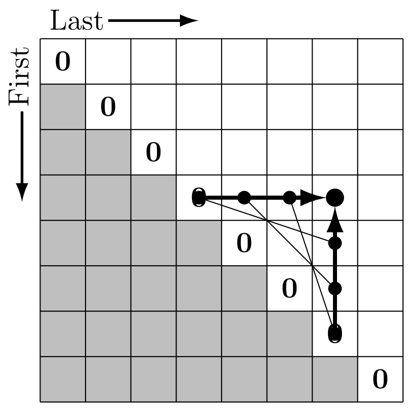
1. Problem: find an optimal parenthesizing of the multiplication of n matrices A_i. Each matrix A_i is of dimension $r_i \times c_i$ and $c_i = r_{i+1} \forall i$. *«r=rows, c=columns»*
2. Matrices: $M_{n\times n}, B_{n\times n}$ *(first, last matrix)*
3. Alphabets:
   - Input: matrix A_i size is defined as pairs of integers (r_i, c_i).
   - Cost matrix: $\Sigma(M) :=$ huge integer[5].
   - Backtrack matrix: $\Sigma(B) = \{stop\} \cup \{0..n\}$.
4. Initialization:
   - Cost matrix: $M_{(i,i)} = 0$.
   - Backtrack matrix: $B_{(i,i)} = stop$.
5. Recurrence: $c_k = r_{k+1}$

$$M_{(i,j)} = \min_{i \le k < j} \left\{ \ M_{(i,k)} + M_{(k+1,j)} + r_i \cdot c_k \cdot c_j \ \middle| \ k \ \right\} = B_{(i,j)}$$

6. Backtracking: Start at $B_{(1,n)}$. Use the following recursive function for parenthesizing

$$\mathrm{BT}(B_{(i,j)} = k) \mapsto \begin{cases} A_i & \text{if } k = 0 \vee k = j \\ \Big(\mathrm{BT}(B_{(i,k)})\Big) \cdot \Big(\mathrm{BT}(B_{(k+1,j)})\Big) & \text{otherwise} \end{cases}$$

7. Visualisation:



8. Optimizations:
   - We need to swap vertically the matrix to have a normalized progression towards bottom right. To do that, we need to map all indices $i \mapsto n-1-i$, but since we want to store the matrix sparsely, we might want to transform it into a lower triangular matrix as we can provide a size-independent mapping of element indices. XXX: what's the best trade off ? progress towards bottom right VS map indices more efficiently ?

[5]Integer multiplication might blow up, use float or doubles and addition of logarithms instead.

2.6 Nussinov algorithm

1. Problem: folding a RNA string S over itself $\lfloor |S|/2 \rfloor = n$.
2. Matrices: $M_{n\times n}, B_{n\times n}$
3. Alphabets:
   - Input: $\Sigma(S) = \{A, C, G, U\}$.
   - Cost matrix: $\Sigma(M) = \{0..n\}$
   - Backtrack matrix: $\Sigma(B) = \{stop, W, S, SW, 1..n\}$
4. Initialization:
   - Cost matrix: $M_{(i,i)} = M_{(i,i-1)} = 0$
   - Backtrack matrix: $B_{(i,i)} = B_{(i,i-1)} = stop$
5. Recurrences (XXX: isn't it $\max_{i\leq k<j}$?):

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

   With $\omega(i,j) = 1$ if i, j are complementary. 0 otherwise.
6. Backtracking: XXX
7. Visualisation: XXX
8. Optimizations: XXX

2.7 Zuker folding

1. Problem: folding a RNA string S over itself $\lfloor |S|/2 \rfloor = n$.
2. Matrices: $M_{n\times n}, V_{n\times n}, B_{n\times n}, F???$
3. Alphabets:
   - Input: $\Sigma(S) = \{A, C, G, U\}$.
   - Cost matrix: $\Sigma(W) = \Sigma(V) = \Sigma(F) = \{0..n\}$
   - Backtrack matrix: $\Sigma(B) = \{stop, S, W, XXXhere\}$
4. Initialization:
   - Cost matrices:
     - XXX: W,V
     - $F_{(0)} = 0$
   - Backtrack matrix: B
5. Recurrence:

$$W_{(i,j)} = \min \left\{ \begin{array}{l|l} W_{(i+1,j)} + b & S \\ W_{(i,j-1)} + b & W \\ V_{(i,j)} + \delta(S_i, S_j) & ?V? \\ \min_{i<k<j} W_{(i,k)} + W_{(k+1,j)} & ?k? \end{array} \right.$$

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

$$F_{(j)} = \min \left\{ \begin{array}{l} F_{(j-1)} \\ \min_{1\le i<j}(V_{(i,j)} + F_{(j-1)}) \end{array} \right. \quad \text{(Free Energy)}$$

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

6. Backtracking: XXX
7. Visualisation: XXX
8. Optimizations: XXX