Dynamic Programming (2/2)

Prof. Rhadamés Carmona

Last revision: 24/01/2020

Content

- Knapsack unbounded
- Matrix chain multiplication
- Longest Common Substring
- Longest Common Sequence
- Distance between strings
- Travelling salesman problem

Knapsack

- Given a set of **n** objects with weight **wi** and value **vi**, and a backpack with **K** capacity, select which objects to take to maximize the value of the items in the bag.
- Bounded: limited amount per type
- Unbounded: unlimited amount per type
- 0/1: There is only one object of each type
- Fractional: you can take a bit of each object

Knapsack unbounded

- A greedy algorithm would sort objects by value/weight ratio. But it's suboptimal.
- The DP version needs a 2D mem.
- Let's define vw obj[0..n-1] as the array of object types, and **int** memo[n,K+1], with memo[i][j] = number of objects selected up to type i, and up to capacity j.
- There are two alternatives for updating memo[i][j]

  Obj[i].w > j: I do not take the i-th object.

  memo[i][j]=memo[i-1][j]

  Obj[i].w <=j: the best option between taking it or not

Knapsack unbounded

- There are two alternatives for updating memo[i][j]
  - Obj[i].w > j: Do not take object i-th
    memo[i][j]=memo[i-1][j].
  - Obj[i].w <=j: best option between taking it or not
    max(Obj[i].v + memo[i][j-Obj[i].w], memo[i-1][j])
- Base cases:
  - $\forall$i memo[i][0] = 0
  - $\forall$j memo[0][j] = obj[0].v * (j / obj[0].w)

Knapsack unbounded

```
struct vw {int v,w; };

int knapsack(vector<vw> &obj, int k)
{
        int i, j, n=obj.size();

        // base cases
        for (i = 0; i < n; i++)  memo[ i ][ 0 ] = 0;
        for (j = 0; j <=k; j++) memo[ 0 ][ j ] = obj[ 0 ].v  * (j / obj[ 0 ].w);
        for (int j = 1; j <= k; j++) for (i = 1; i < n; i++)
          if (j < obj[ i ].w)  // objet i does not fits in a bag with capacity j
            memo[ i ][ j ] = memo[ i – 1 ][ j ];
          else   // it fits, but let's take the best option
            memo[ i ][ j ] = max(obj[ i ].v + memo[ i ][ j-obj[ i ].w ], memo[ i-1] [ j ]);
        return memo[n-1][k];  // total value (we can also store total weigh)
}
```

Knapsack unbounded

- When memo[i,j]==memo[i-1,j] then we did not take the i-th object at this point of the solution (continue with --i); otherwise, we take the i-th object and we continue checking with
  memo[i, j-object[i].weight]
- Check the pairs (ellipsoids) to build the winner solution (see next slide)

Knapsack unbounded

v/w	0	1	2	3	4	5	6	7	8	9	10
2/**4**	0	0	0	0	2	2	2	2	4	4	4
3/**2**	0	0	3	3	6	6	9	9	12	12	15
5/**3**	0	0	3	5	6	8	10	11	13	15	16

Two with value 5 and weight 3, two with value 3 and weight 2 (tot value = 16, tot weight = 10)

Knapsack unbounded

v/w	0	1	2	3	4	5	6	7	8	9	10
1/**2**	0	0	1	1	2	2	3	3	4	4	5
4/**3**	0	0	1	4	4	5	8	8	9	12	12
7/**4**	0	0	1	4	7	7	8	11	14	14	15

We do not need to fill the bag to the top in some cases

One with v=7 and w=4, two with v=4 and w=3 (Total v=15, Total w=10)

Matrix chain multiplication

- We want to multiply matrices A0*A1*...*An-1
- The question is: how do we apply associative property to minimize the number of products.
- We know that Cn,m = An,k*Bk,m requires n*k*m multiplications.
- Example: A4.2*B2.8*C8.1
- requires 4*2*8+4*8*1=96 when performing (A*B)*C ➔ first A*B
- Requires 2*8*1+4*2*1=24 when performing A*(B*C) ➔ first B*C

Matrix chain multiplication

- Supose that we store in memo[i,j] the minimal number of operations to multiply Ai*Ai+1*…*Aj-1*Aj, with sizes $(n_i * n_{i+1}) \times (n_{i+1} * n_{i+2}) \times \ldots \times (n_j * n_{j+1})$
- memo[i,j] can be computed as the minimum between
  - (Ai)*(Ai+1*Ai+2*…* Aj-1*Aj)

    $0 + (n_i * n_{i+1} * n_{j+1})$ + memo[i+1,j]
  - (Ai* Ai+1)*(Ai+2 *Ai+3…* Aj-1*Aj)

    memo[i,i+1]+$(n_i * n_{i+1} * n_{i+1})$+memo[i+2,j]

    …
  - (Ai*Ai+1*…*Ak-1)* (Ak*…*Aj)

    Memo[i,k-1]+$(n_i * n_k * n_{j+1})$ +memo[k,j]

    …
  - (Ai*Ai+1*Ai+2……Aj-2*Aj-1)*(Aj)

    Memo[i,j-1]+$(n_i * n_j * n_{j+1})$+0

Matrix chain multiplication

- Base case:
  - Memo[i,i]=0
  - Memo[i,i+1]=$n_i * n_{i+1} * n_{i+2}$
- Cost of combining 2 results given i, k, j:
  - (Ai*…*Ak-1)*(Ak*…*Aj):

  comb(i,k,j) = $n_i * n_k * n_{j+1}$
- memo[i,j] =
  - min{ memo[i,k-1] + comb(i,k,j) + memo[k, j],

    with k=i+1,...,j }

Matrix chain multiplication

- We use half of the 2D Memo array
- The result is found in Memo[0][m-1]

Matrix chain multiplication

```
int matrixChainMult(vector <int> &n) {  // test with n={4,2,8,1}, A_{4,2}*B_{2,8}*C_{8,1}
        int m = n.size() - 1;
        for (int i = 0; i < m; i++) {
                memo[i][i] = 0;
                memo[i][i +1] = n[i] * n[i+1] * n[i + 2];
        }
        for (int i = 0; i < m; i++) for (int j = i+2; j < m; j++) {
                int min = INFINITE;
                for (int k = i+1; k < j; k++) {
                    int val memo[i][k-1] + n[i]*n[k]*n[j+1] + memo[k][j];
                    if (val < min) min = val;
                }
                memo[i][j] = min;
        }
        return memo[0][m - 1];
}
```

Matrix chain multiplication

- Homework: Rebuild the winner solution
- Print the winner solution using full parenthesis, e.g.

(A0 * A1) * ((A2) * (A3 * A4))

LCS:
Longest Common Substring

- We have two a and B strings of size n and m
- The "naive" solution is to start on each pair of possible boxes A[i], B[j] and check character by character as long as they match, which is cubic order.
- The DP solution uses a 2D array of n*m boxes where memo[i,j] contains the length of the match ending at the i,j (inclusive).

LCS:
Longest Common Substring

- TO compute memo[i,j] we have 2 possibilities
  - A[i]==B[j]: memo[i,j] = 1 + memo[i-1,j-1]
  - A[i]!=B[j]: memo[i,j] = 0
- Base cases
  - Memo[0,j] = (A[0]==B[j]) ?1:0
  - Memo[i,0] = (A[i]==B[0]) ?1:0
- Every time we update a cell, update the global maximum if possible

LCS:
Longest Common Substring

```
void buildmatrix(const string &a, const string &b)
{
        int n = a.size();
        int m = b.size();
        for (int i = 0; i < n; i++)
            memo[i][0] = (a[i] == b[0]);
        for (int j = 0; j < m; j++)
            memo[0][j] = (a[0] == b[j]);
        for (int i = 1; i < n; i++)
            for (int j = 1; j < m; j++)
                memo[i][j] = (a[i] == b[j]) ? 1 + memo[i - 1][j - 1] : 0;
}
```

LCS:
Longest Common Substring

```
position getLongest(int n, int m)
{
        int I = 0, J = 0, V = -INFINITE;
        for (int i = 0; i < n; i++) for (int j = 0; j < m; j++)
                if (memo[i][j] > V)
                {
                        V = memo[i][j];
                        I= i;
                        J = j;
                }
        return position(I,J,V);
}
```

The solution substring is obtained from A[I], or B[J], by traversing V positions to the left.
If there are multiple solutions, how to:
- Get all the solutions?
- Get the minor lexicographically?

LCS:
Longest Common Sequence

- In this case, there could be “jumps”; then, it may not be a consecutive sequence
- Example: A=**abc**d**ef**g**h**i, B=**abc**ac**efh**x, **abcefh** is the solution with length 6
- There are 2 cases:
  - A[i]=B[j]: Memo[i,j] = 1+ Memo[i-1,j-1]
  - A[i]$\neq$B[j]: Memo[i,j] =

    Max { Memo[i-1,j], Memo[i,j-1] }

LCS:
Longest Common Sequence

- It means, if they are different we search the best solution ignoring the i-th element or the j-th element. Ignoring both at the same time does not generate a better solution than ignoring only one.
- Base case
  - Memo[0,j] = (A[0]==B[j]) ?1:0
  - Memo[i,0] = (A[i]==B[0]) ?1:0

LCS:
Longest Common Sequence

```
void buildmatrix(const string &a, const string &b)
{
        int n = a.size();
        int m = b.size();
        for (int i = 0; i < n; i++)
          memo[i][0] = (a[i] == b[0]);
        for (int j = 0; j < m; j++)
          memo[0][j] = (a[0] == b[j]);
        for (int i = 1; i < n; i++)
          for (int j = 1; j < m; j++)
              memo[i][j] = (a[i] == b[j]) ? 1 + memo[i - 1][j - 1] :
                                 std::max (memo[i-1][j], memo[i][j-1] );
}
```

Distance between strings

- How many changes need to be made for one string to be the same as another.
- Unitary operations: add character, delete character, replace character.
- Examples: between hou**s**e and hou**z**e it is enough to replace **z** by **s** (one operation). Between hello and **y**ello**w**, replace **y** by **h**, and remove **w** (two operations).

Distance between strings

- n=A.size()   m=B.size()
- For updating memo[i][j]:
  - n==0: m
  - m==0: n
  - A[i]==B[j]: memo[i-1][j-1]
  - A[i]!=B[j]:
    - Case 1: Set A[i] = B[j] → 1+memo[i-1][j-1]
    - Case 2: Delete A[i] → 1+memo[i-1][j]
    - Case 3: Insert B[j] after A[i] → 1+memo[i][j-1]

Distance between strings

```
int stringDistance(const string &a, const string &b)
{
        int n = a.size(), m = b.size();
        for (int i = 0; i < n; i++) memo[i][0] = i;
        for (int j = 0; j < m; j++)memo[0][j] = j;
        for (int i = 1; i < n; i++) for (int j = 1; j < m; j++)
                if (a[i] == b[j])
                    memo[i][j] = memo[i - 1][j - 1];
                else
                    memo[i][j] = 1+ min(min(memo[i - 1][j],
                                    memo[i][j - 1]), memo[i - 1][j - 1]);
        return memo[i][j];
}
```

Travelling salesman problem

- Given an undirected weighted graph representing connections between cities, find a tour that visit all the cities only once.
- The brute force solution would permutate the n cities, and verify the cost of each "feasible" permutation, keeping the one with lowest total cost (n! possibilities)

Travelling salesman problem

- DP solution: supose that the number of cities is <= 64. Thus, we can represent the active cities with a single integer number or an array of booleans.
- Let **c** be an integer representing the current set of active cities (each “on” bit represents an unvisited city), and **w**(**i**,**j**) = weight of Edge (**i**,**j**)
- F(**c**, **i**) = min { F(**c** – (1<<**i**), **j**) + **w**(**i**,**j**) , for every **j**!=**i** such that **c**&(1<<**j**) != 0 && **w**(**i**,**j**)!=0}
- F(0,**i**) = 0 (base case)

Travelling salesman problem

- Because **c** can take 2^n possible values, the table might not be representable in memory, so you can use a hash table or a map.
- In the worst case, the minimum per box is taken between n options, and the table is $n.2^n$ ➔ complexity is $O(n^2 2^n)$ which is asymptotically less than n!, but still exponential.

DP Problems

- https://www.spoj.com/problems/DBALLZ/
- https://www.spoj.com/problems/KNAPSACK/
- https://www.spoj.com/problems/MIXTURES/cstart=40
- https://www.spoj.com/problems/LISA/
- https://www.spoj.com/problems/ADFRUITS/
- https://www.spoj.com/problems/LCS/
- https://www.spoj.com/problems/LCS0/

DP Problems

- https://www.spoj.com/problems/SAMER08D/
- https://www.spoj.com/problems/EDIST/
- https://www.spoj.com/problems/STRDIST/
- https://www.spoj.com/problems/ADVEDIST/
- https://www.spoj.com/problems/STSP/
- https://www.spoj.com/problems/PESADA04/

DP Problems

- Implement at least one problem of each type
- Knapsack
- MCM
- LCSe
- LCSu
- TSP
- Another DP problem you find challenging and different