Dynamic Programming

1. Definition

Dynamic programming is a mathematical method that breaks down a problem into subproblems so that :

- •1 subproblem optima can be combined to give a solution to the original problem, known as *optimal substructure*
- •2 subproblem optima follows *Bellman optimality equation* which governs minimum total cost starting from state s_0

$$
\begin{aligned}
f(s_0) &= \min_{a_0 a_1 a_2 \ldots \in action} \left[\sum_{t\geq 0} cost(s_t, a_t)\right] \\
&= \min_{a_0 \in action} \left[cost(s_0, a_0) + \min_{a_1 a_2 \ldots \in action} \sum_{t\geq 1} cost(s_t, a_t)\right] \\
&= \min_{a_0 \in action} \left[cost(s_0, a_0) + f(s_1)\right]
\end{aligned}
$$

objective is sum of cost, optimal objective is f

- •3 implemented as iteration if subproblems overlap, such as Fibonacci sequence
  implemented as iteration or recursion if subproblems do not overlap, such as factorial
- •4 some problems can be represented as *subproblem-graph*, some can be represented as *state-time-graph* (my terminology)
- for subproblem-graph, row index is *n*, while column index is *m*
- for state-graph, row index is *constrainted state C*, while column index is *time t*

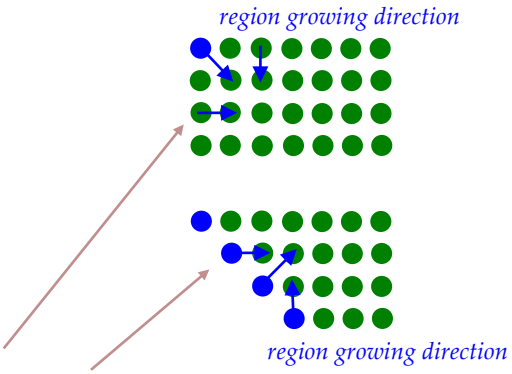
2. Subproblem-graph

Subproblem-graph *G = {V,E}* denotes the relationship among subproblem optima :

V = *{ value function of problems with different sizes }*
E = *{ relation between two subproblems optima }*

$$
G = \begin{array}{c} \\ n\ axis \downarrow \end{array}
\begin{array}{c} m\ axis \rightarrow \\ \end{array}
\begin{bmatrix}
f(1,1) & f(1,2) & f(1,3) & \ldots & f(1,M) \\
f(2,1) & f(2,2) & f(2,3) & \ldots & f(2,M) \\
f(3,1) & f(3,2) & f(3,3) & \ldots & f(3,M) \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
f(N,1) & f(N,2) & f(N,3) & \ldots & f(N,M)
\end{bmatrix}
$$

subproblem graph for 2D problem



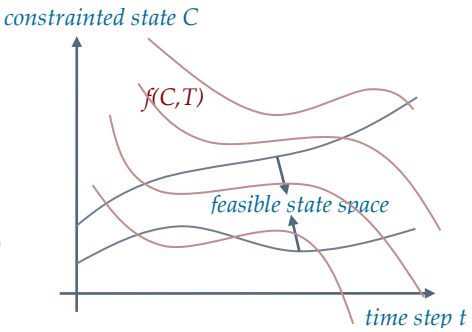
2D problem is usually related to :

- combinatorial search between two vectors `x[1:N]` and `y[1:M]` with size `N` and `M` or
- combinatorial search within one vector for its start and end `x[N:M]` where `N < M < x.size`
- these two problems will end up with different graphs :
- -1 the former results in a matrix with region growing started from origin *(such as longest common subsequence)*
- -2 the latter results in a UL matrix with region growing started from diagonal *(such as parenthesization and optimal game)*

3. State-time-graph

State-time-graph *G = {V,E}* on the contrary, denotes the transition between states and corresponding cost :

V = *{ value function v = f(C,T) of constrainted state C when T time steps }*
E = *{ cost for transition between two states }*



- if it is a multipartite-graph, it can be solved for ***Viterbi***
- if it is a directed acyclic graph, can be solved by ***Dijkstra***
- ***Viterbi*** is region growing using two vectors `v0 = std::move(v1)`
- ***Dijkstra*** is region growing using priority queue

The **seven elements** for state-graph problem *(making change, knapsack and job scheduling)*

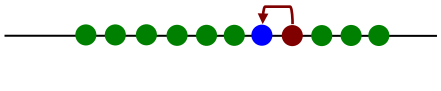
- •1,2 *set of N choices with cost-value pairs* $[(x_1,y_1), (x_2,y_2), (x_3,y_3), \ldots]$
- •3 *set of N decision* $K = [k_1, k_2, k_3, \ldots]$
- •4 *update decision vector K once every time step* $t \in [1, \infty]$
- •5 *optimize objective* $\sum_{n\in[1,N]} k_n y_n$
- •6 *under constraint* $\sum_{n\in[1,N]} k_n x_n = C$
- •7 *Bellman solution* $f(C,T) = \underset{k}{opt} \sum_{n\in[1,N]} k_n y_n$ *such that* $\sum_{n\in[1,N]} k_n x_n = C$ *note that f is a function of C and T*

4. Various problems 231771

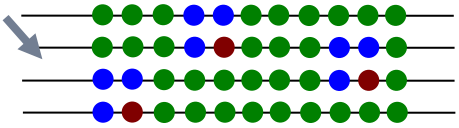
Subproblem-graph

Single-link recursion on list

- Maximum subsequence sum
- *L* shape block



region growing direction

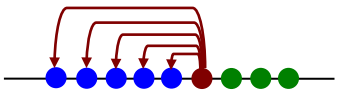


Single-link recursion on matrix

- Longest common subsequence `diagram /rec/bc/subpg`
- Building bridge
- Edit distance `game-tree/rec/bc/subpg`

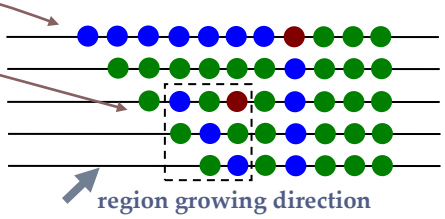
Multi-links recursion on list

- Bell number



Multi-links recursion on matrix or cube

- Boolean parenthesization `game-tree/rec/bc/subpg`
- Two coin game `game-tree/rec/bc/subpg`
- Two persons traversal of cities `game-path/rec/bc/subpg`
- Bitonic tour `game-path/rec/bc/subpg`
- Piecewise linear `diagram /rec/bc/subpg`
- Box stacking
- Bin packing ← *breakdown into 2 subproblems*



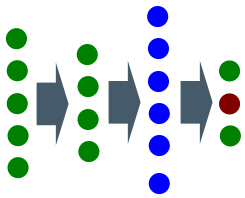
region growing direction

State-time-graph

Multi-steps recursion on multipartite graph *(region growing in time domain)*

- Viterbi algorithm
- Retrospective trader
- Assembly line pair
- Change making problem
- Knapsack problem
- Job scheduling
- Equal partition sum

fully connected multipartite graph



Multi-steps recursion on directed acyclic graph *(region growing using priority queue)*

- Dijkstra algorithm

Remark

```
rec   = recursive function
bc    = boundary condition
subpg = subproblem graph
```

Maximum subsequence sum ***MSS***

Given *N* numbers, find the maximum subsequence sum.

- Brute force search is $O(N^3)$, can be speeded up to $O(N^2)$ with the help of cumulative sum.
- Dynamic programming (called Kadane algorithm) can be done in $O(N)$.
- Divide and conquer can be done in $O(NlogN)$.

Exhaustive search in $O(N^3)$

```
int max_subseq_sum(const std::vector<int>& x)
{
     int ans = x[0];
     for(int n=0; n!=x.size(); ++n)
     {
          for(int m=n+1; m!=x.size(); ++m)
          {
               int cum = std::accumulate(x.begin()+n, x.begin()+m+1);
               ans = std::max(ans, cum);
          }
     }
     return ans;
}
```

Exhaustive search in $O(N^2)$

It is just a 2D scan (with `n` as starting point, `m` as ending point) keep check of cumulative sum and maximum sum.

```
int max_subseq_sum(const std::vector<int>& x)
{
     int ans = x[0];
     for(int n=0; n!=x.size(); ++n)
     {
          int cum = x[n];
          for(int m=n+1; m!=x.size(); ++m)
          {
               cum += x[m];
               ans = std::max(ans, cum);
          }
     }
     return ans;
}
```

Dynamic programming in O(N)

Subproblem `n` is defined as maximum sum subsequence **ending in** `x[n]`, we have to derive suboptimum `n` from suboptimum `n-1`.

- suboptimum `n-1` must : start with `x[m]` and ends with `x[n-1]`, where `m <= n-1`
- suboptimum `n` must either : start with the same `x[m]` and ends with `x[n]`, where `m <= n-1`
  or start with `x[n]` and ends with `x[n]` (if it starts with `x[m]` then it must fall into previous case)

```
int max_subseq_sum(const std::vector<int>& x)
{
     int ans = x[0];
     int sub = x[0];
     for(int n=1; n!=x.size(); ++n)
     {
          sub = std::max(sub+x[n], x[n]);
          ans = std::max(ans, sub);
     }
     return ans;
}
```

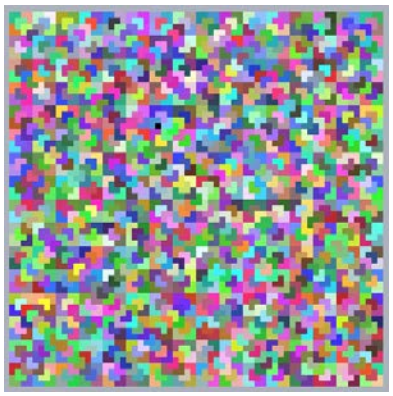
Divide and conquer in O(NlogN)

Divide the vector into two halves, the maximum is one of the three cases (1) both starting and ending points in the first half (2) both starting and end point in the seocond half (3) starting point in the first half and ending point in the second half.

```
int max_subseq_sum(const std::vector<int>::iterator begin, const std::vector<int>::iterator end)
{
     auto mid = (begin+end)/2;
     int cum0 = x[mid-1];  int max0 = cum0;
     int cum1 = x[mid];    int max1 = cum1;

     for(auto iter=mid-2; iter >= begin; --iter) { cum0 += *iter; max0 = std::max(max0, cum0); }
     for(auto iter=mid+1; iter <  end;   ++iter) { cum1 += *iter; max1 = std::max(max1, cum1); }
     return std::max(max0+max1, max_subseq_sum(begin,mid) , max_subseq_sum(mid,end));
}
```

L Shape Block

Given a 2^N by 2^N grid, a pixel is randomly picked, find an algorithm to fill the whole grid by *L* shape blocks, except for chosen pixel.



Solution

Lets demonstrate the recursion in an algorithmic way.

```
bitmap f(int N, int y, int x) // Use (y,x) for matrix coordinate, instead of (x,y).
{
     if (N==1) return fill_single_L(y,x);

     int y0 = 2^(N-1)-1;          int x0 = 2^(N-1)-1;
     int y1 = 2^(N-1)-1;          int x1 = 0;
     int y2 = 0;                  int x2 = 2^(N-1)-1;
     int y3 = 0;                  int x3 = 0;

     if (y < 2^(N-1) && x < 2^(N-1)) { y0 = y;           x0 = x;           }
     if (y < 2^(N-1) && x > 2^(N-1)) { y1 = y;           x1 = x-2^(N-1);   }
     if (y > 2^(N-1) && x < 2^(N-1)) { y2 = y-2^(N-1);   x2 = x;           }
     if (y > 2^(N-1) && x > 2^(N-1)) { y3 = y-2^(N-1);   x3 = x-2^(N-1);   }

     lower_range = 0 : 2^(N-1);           // begin and end (use std::vector convention)
     upper_range =     2^(N-1) : 2^N;     // begin and end
     bitmap(lower_range, lower_range) = f(N-1, y0, x0);
     bitmap(lower_range, upper_range) = f(N-1, y1, x1);
     bitmap(upper_range, lower_range) = f(N-1, y2, x2);
     bitmap(upper_range, upper_range) = f(N-1, y3, x3);
     return bitmap;
}
```

Longest common subsequence

Given N symbols $x_1, x_2, \ldots x_N$ and M symbols $y_1, y_2, \ldots y_M$, find the longest common subsequence, which isn't necessarily contiguous. Software *winmerge* is an example of longest common subsequence.

Solution

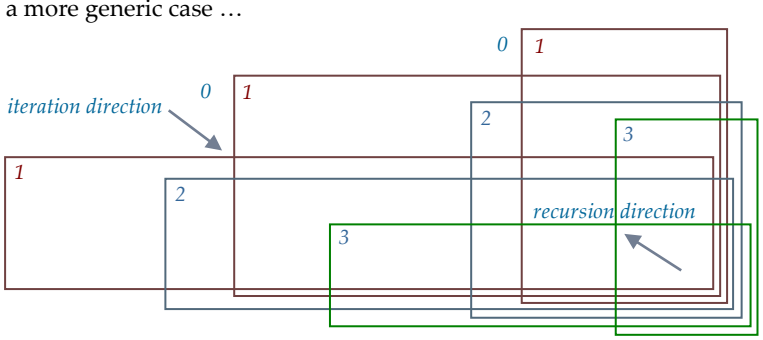
Suppose *f(N,M)* is length of the longest common subsequence, which is a 2*D* value function. Let's consider :

- string `"A...B...C...D"` versus
- string `"C...D...A...B"` where `...` denotes a unique, impossible-to-match sequence.

	y[:]= C	y[:]= CD	y[:]= CDA	y[:]= CDAB
x[:]= A	0	0	**A**	A
x[:]= A...	0	0	A	A
x[:]= AB	0	0	A	**AB**
x[:]= AB...	0	0	A	AB
x[:]= ABC	**C**	C	A/C	AB
x[:]= ABC...	C	C	A/C	AB
x[:]= ABCD	C	**CD**	CD	AB/CD
x[:]= ABCD...	C	CD	CD	AB/CD

We fill the *f(N,M)* matrix using row-by-row scan. We observe that as we scan …

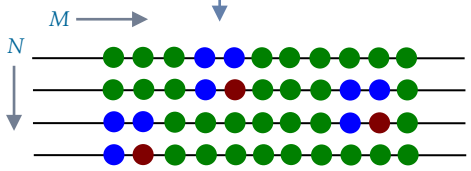
- when `x[N]==y[M]`, i.e. all the **red-bolded-entries**, there is an increment in common length by one
- when `x[N]!=y[M]`, i.e. all other entries, there must be no increment in common length
- each **red-bolded-entry** spreads its value to LR corner (infinity) as denoted by the colored rectangles
- when an entry is covered by more than one rectangle, it simply takes the maximum value
- here is a more generic case …



(1) game diagram
(2) recursion equation
(3) boundary condition
(4) subproblem graph

$$f(N,M) = \begin{cases} f(N-1,M-1)+1 & \text{if } x_N = y_M \\ \max(f(N-1,M), f(N,M-1)) & \text{if } x_N \neq y_M \end{cases} \quad \text{recursion}$$

$$f(1,1) = \begin{cases} 1 & \text{if } x_1 = y_1 \\ 0 & \text{if } x_1 \neq y_1 \end{cases} \quad \text{boundary}$$



Here is the subproblem graph that related *f(N,M)* with its 3 neighbours in UL corner.

Implementation can be done by :

- iterative approach, which scans the matrix horizontally / vertically / diagonally, invokes recursion equation ←
- recursive approach, which does not scan the matrix, invokes recursion equation →
- recursive approach involves seriously overlapping subproblems, it is much slower

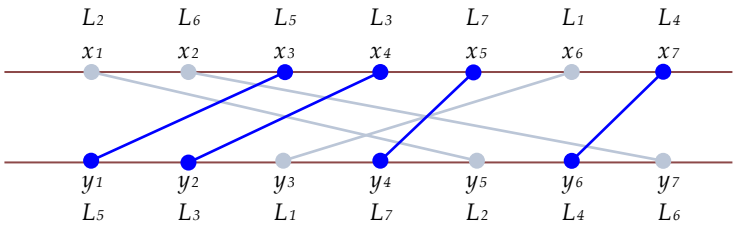
```
int LCS_by_iteration(const std::vector<T>& x, const std::vector<T>& y) // C++ vector starts with index0
{
    // origin
    if (x[0]==y[0]) f(0,0)=1; else f(0,0)=0;

    // scan 1st row
    for(m=1; m!=M; ++m) { if (x[0]==y[m]) f(0,m) = 1; else f(0,m) = f(0,m-1); }

    // scan 1st col
    for(n=1; n!=N; ++n) { if (x[n]==y[0]) f(n,0) = 1; else f(n,0) = f(n-1,0); }

    // scan matrix
    for(n=1; n!=N; ++n)
    {   for(m=1; n!=M; ++m)
        {
            if (x[n]==y[m])    f(n,m) = f(n-1,m-1)+1;
            else               f(n,m) = std::min(f(n,m-1), f(n-1,m));
        }
    }
    return f(N-1,M-1);
}
```

Building bridge

Given N and M cities respectively lying along the north and south coast of a horizonal river, having x-coordinates : $x_1, x_2, \ldots x_N$, and and $y_1, y_2, \ldots y_N$. Suppose $K = max(N,M)$ unique labels $L_1, L_2, L_3, \ldots L_K$ are randomly assigned to both the cities on the north coast and the south coast, cities on the same coast must have different labels. We can build bridges across the river joining cities having same label. What is the maximum number of bridges that we can build?

Solution

If we sort the cities by x-coordinates, and draw a line joining cities with the same label, we have for example :



Maximizing the number of bridges without crossing each other is a special case of *longest common subsequence*, in which all elements in the sequence (labels, not the coordinates) are unique. We sort labels by coordinates, coordinates are irrelevant to the solution.

Edit distance

Given two strings, $\{x_1,x_2,x_3,\ldots,x_N\}$ and $\{y_1,y_2,y_3,\ldots,y_M\}$, having length N and M, edit distance is then defined as the minimum number of operations needed to convert string A into string B, providing that there are 3 different valid operations :

- insert a character into string x
- delete a character in string x
- modify a character of string x

Solution

Let $f(N,M)$ be the edit distance between string $\{x_1,x_2,x_3,\ldots,x_N\}$ and string $\{y_1,y_2,y_3,\ldots,y_M\}$, we then have :

	original string		*intermediate*		*output string*	
case 1 $x_N = y_M$	$\{x_1x_2x_3...x_{N-1}x_N\}$	$\xrightarrow{f(N-1,M-1)}$	$\{y_1y_2y_3...y_{M-1}x_N\}$	$=$	$\{y_1y_2y_3...y_{M-1}y_M\}$	*since* $x_N = y_M$
case 2 $x_N \neq y_M$	$\{x_1x_2x_3...x_{N-1}x_N\}$	$\xrightarrow{f(N,M-1)}$	$\{y_1y_2y_3...y_{M-2}y_{M-1}\}$	$\xrightarrow{insert(y_M)}$	$\{y_1y_2y_3...y_{M-1}y_M\}$	
	$\{x_1x_2x_3...x_{N-1}x_N\}$	$\xrightarrow{f(N-1,M)}$	$\{y_1y_2y_3...y_{M-1}y_Mx_N\}$	$\xrightarrow{delete(x_N)}$	$\{y_1y_2y_3...y_{M-1}y_M\}$	
	$\{x_1x_2x_3...x_{N-1}x_N\}$	$\xrightarrow{f(N-1,M-1)}$	$\{y_1y_2y_3...y_{M-1}x_N\}$	$\xrightarrow{\text{mod}ify(x_N \to y_M)}$	$\{y_1y_2y_3...y_{M-1}y_M\}$	

Thus we have a recursion very similar to longest common subsequence.

$$f(N,M) = \begin{cases} f(N-1,M-1) & \text{if } x_N = y_M \\ \min(f(N-1,M)+1, f(N,M-1)+1, f(N-1,M-1)+1) & \text{if } x_N \neq y_M \end{cases} \quad \text{recursion equation}$$

$$f(1,M) = M-1(x_1 \in \{y_1,y_2,y_3,...y_M\}) \quad \text{boundary condition}$$

$$f(N,1) = N-1(y_1 \in \{x_1,x_2,x_3,...x_N\}) \quad \text{boundary condition}$$

The implementation is done in same way too.

Bell number

Bell number is the number of ways to partition N elements.

Solution

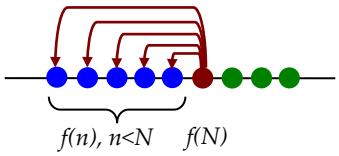
Let's try a simpler problem : the number of combinations for N elements. It is done by :

$$\begin{aligned}
\textit{picking zero element} &= C_0^N \\
\textit{picking one element} &= C_1^N \\
\textit{picking two elements} &= C_2^N \\
\dots & \\
\textit{picking N elements} &= C_N^N \\
\textit{total(N)} &= \sum_{n=1}^{N} C_n^N = 2^N
\end{aligned}$$

Combination does not care about the arrangement of the elements left behind. Now suppose $f(N)$ is Bell number. Element N should belong to one partition, what we are going to do is to pick and place n extra elements into the partition element N lies then partition the rest by recursion of $N\text{-}1\text{-}n$ elements, which is $f(N\text{-}1\text{-}n)$. It is done by :

$$\begin{aligned}
\textit{picking zero element} &= C_0^{N-1} f(N-1) \\
\textit{picking one element} &= C_1^{N-1} f(N-2) \\
\textit{picking two elements} &= C_2^{N-1} f(N-3) \\
\dots & \\
\textit{picking N-1 elements} &= C_{N-1}^{N-1} f(0) \\
f(N) &= \sum_{n=0}^{N-1} C_n^{N-1} f(N-1-n) \\
&= \sum_{m=N-1}^{0} C_{N-1-m}^{N-1} f(m) \qquad \textit{putting } m = N-1-n \\
&= \sum_{m=0}^{N-1} C_m^{N-1} f(m) \qquad \textit{since } C_r^N = C_{N-r}^N \\
\textit{boundary case } f(0) &= 1
\end{aligned}$$

Here is the subproblem-graph for $f(N)$, it depends on all $f(n)$ where $n<N$:



Here is an example of Bell number up to 4. The *red highlighted* items represent the partition where element N lies.

$f(0) = 1$ $()$

$f(1) = 1$ (x_1)

$f(2) = 2$ $(x_2)(x_1)$ (x_2x_1)

$f(3) = 5$ $(x_3)(x_2)(x_1)$ $(x_3)(x_2x_1)$ $(x_3x_2)(x_1)$ $(x_3x_1)(x_2)$ $(x_3x_2x_1)$

$f(4) = 15$ $(x_4)(x_3)(x_2)(x_1)$ $(x_4)(x_3)(x_2x_1)$ $(x_4)(x_3x_2)(x_1)$ $(x_4)(x_3x_1)(x_2)$ $(x_4)(x_3x_2x_1)$ $(x_4x_3)(x_2)(x_1)$ $(x_4x_3)(x_2x_1)$ $(x_4x_2)(x_3)(x_1)$ $(x_4x_2)(x_3x_1)$ $(x_4x_1)(x_3)(x_2)$ $(x_4x_1)(x_3x_2)$ $(x_4x_3x_2)(x_1)$ $(x_4x_3x_1)(x_2)$ $(x_4x_2x_1)(x_3)$ $(x_4\ x_3\ x_2\ x_1)$

Boolean parenthesization

Given an expression with *N* binary operators (hence *N+1* operands), find the number of different parenthesizations that give *TRUE* as the final result, providing that (1) there are *3* possible operators : *AND*, *OR*, *XOR* (recall that *XOR* returns *TRUE* when inputs are different), (2) there are 2 possible operands : *TRUE* and *FALSE*. We assume no default operator precedence, the priority of operator execution depends only on parenthesization, hence bracket is needed to resolve the priority between operators.

$x_1\ op_1\ x_2\ op_2\ x_3\ op_3\ \dots\ x_N\ op_N\ x_{N+1}$

Solution

We can observe that a general expression has *N+1 operands*, *N operators*, *N-1 brackets* are needed.

N	*#operand*	*#bracket*	*parenthesization*	*# parenthesization*
0	1	0	x_1	1
1	2	0	$x_1\ op_1\ x_2$	1
2	3	1	$x_1\ op_1\ (x_2\ op_2\ x_3)$	$1\times1+1\times1=2$
			$(x_1\ op_1\ x_2)\ op_2\ x_3$	
3	4	2	$x_1\ op_1\ (x_2\ op_2\ (x_3\ op_3\ x_4))$	$1\times2+1\times1+2\times1=5$
			$x_1\ op_1\ ((x_2\ op_2\ x_3)\ op_3\ x_4)$	
			$(x_1\ op_1\ x_2)\ op_2\ (x_3\ op_3\ x_4)$	
			$(x_1\ op_1\ (x_2\ op_2\ x_3))\ op_3\ x_4$	
			$((x_1\ op_1\ x_2)\ op_2\ x_3)\ op_3\ x_4$	

Parenthesization is equivalent to conversion of an expression into a binary tree, with :

- each operand x_n is a leaf node
- each operator as a parent node
- innermost-bracketed operator denotes branching near tree bottom
- outermost-unbracketed operator denotes branching near tree top (i.e. *red-highlighted* unbracketed-operator in above example)
- the number of parenthesization means the number of possible tree that gives true / false at the root

Change in convention, please adopt the convention as in the code next page : (I cant do that now, as word 2019 does not support equation)

- *replace N below by* `n`, *where* `n=[0,N]`
- *replace M below by* `m`, *where* `m=[0,N]`
- *replace n below by* `k`

Define

$$f(N,M) = num_{True}[x_N \cdot op_N \cdot x_{N+1} \cdot op_{N+1} \cdot x_{N+2} \cdots x_{M-1} \cdot op_{M-1} \cdot x_M] \quad \text{where } N \le M$$

$$g(N,M) = num_{False}[x_N \cdot op_N \cdot x_{N+1} \cdot op_{N+1} \cdot x_{N+2} \cdots x_{M-1} \cdot op_{M-1} \cdot x_M] \quad \text{where } N \le M$$

Recursion is :

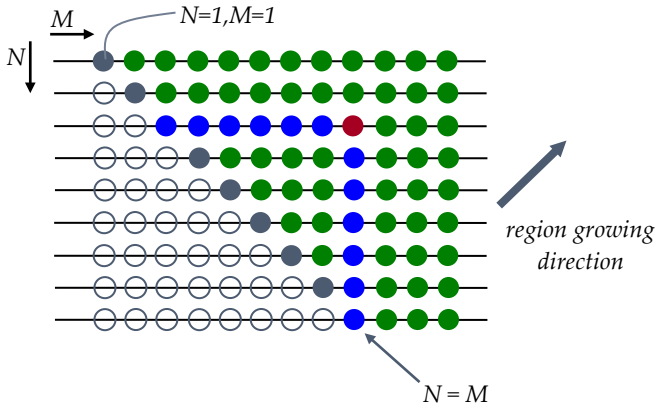
$$f(N,M) = \sum_{n=N}^{M-1} \begin{bmatrix} 1(op_n = and)\times[f(N,n)f(n+1,M)] + \\ 1(op_n = xor)\times[f(N,n)g(n+1,M) + g(N,n)f(n+1,M)] + \\ 1(op_n = or)\times[f(N,n)g(n+1,M) + g(N,n)f(n+1,M) + f(N,n)f(n+1,M)] \end{bmatrix}$$

$$g(N,M) = \sum_{n=N}^{M-1} \begin{bmatrix} 1(op_n = and)\times[g(N,n)g(n+1,M) + g(N,n)f(n+1,M) + f(N,n)g(n+1,M)] + \\ 1(op_n = xor)\times[g(N,n)g(n+1,M) + f(N,n)f(n+1,M)] + \\ 1(op_n = or)\times[g(N,n)g(n+1,M)] \end{bmatrix}$$

Boundary case is :

$$f(N,N) = \begin{bmatrix} 1 & x_N = true \\ 0 & x_N = false \end{bmatrix}$$

$$g(N,N) = \begin{bmatrix} 0 & x_N = true \\ 1 & x_N = false \end{bmatrix}$$



RHS is subproblem graph :

- red node *f(N,M)* depends on blue nodes *f(N,N:M-1)* and *f(N+1:M,M)*
- empty nodes denote invalid state
- it is liked region growing from diagonal toward UR corner

The implementation involves 3 layer for loops.

```
int num_parenthesis(const std::vector<bool>& x, const std::vector<OP>& op)
{
    if (x.size()!=op.size()+1) return std::make_pair(0,0);
    square_matrix f(op.size()+1); f.reset_all(0);
    square_matrix g(op.size()+1); g.reset_all(0);

    // Initialize diagonal
    for(int n=0; n!=op.size()+1; ++n)
    {
        if (x[n]) { f(n,n) = 1;  g(n,n) = 0; }
        else      { f(n,n) = 0;  g(n,n) = 1; }
    }

    // Useful pattern : Region growing from diagonal
    for(int d=1; d<=op.size(); ++d) // d means the d-th diagonal
    {
        for(int x=0; x!=op.size()-(d-1); ++x) // x means the x-the element in d-th sub-diagonal
        {                                     // there are op.size()   elements in 1st sub-diagonal
            // Transform (d,x) to (n,m)       // there are op.size()-1 elements in 2nd sub-diagonal ... and so on
            int n = x;
            int m = x+d;

            for(int k=n; k!=m; ++k) // scan two legs
            {
                if (op[k]==AND)      { f(n,m) += f(n,k)*f(k+1,m);
                                       g(n,m) += f(n,k)*g(k+1,m) + g(n,k)*f(k+1,m) + g(n,k)*g(k+1,m); }
                else if (op[k]==XOR) { f(n,m) += f(n,k)*g(k+1,m) + g(n,k)*f(k+1,m);
                                       g(n,m) += f(n,k)*f(k+1,m) + g(n,k)*g(k+1,m);                   }
                else                 { f(n,m) += f(n,k)*f(k+1,m) + f(n,k)*g(k+1,m) + g(n,k)*f(k+1,m);
                                       g(n,m) += g(n,k)*g(k+1,m);                                     }
            }
        }
    }
    return f(0,op.size());
}
```

Two coin games – Game 1

This is a question I encountered in the interview with Hashkey.com on 5 June 2019. This is not about dynamic programming. Given *N* coins, you and your opponent are requested to remove *1-5* coins strategically in turn whichever party gets the last coin wins, you are the party to make the first move, what will be your strategy?

Solution

Suppose you and your opponent are equally rational. Now instead of attempting dynamic programming, we try iterative approach. Let's define *N* as the problem size and *f(N)* as the number of coins to be removed in the first round of an optimal strategy, thus after your first move, you are leaving a subproblem *f(N-f(N))* to your opponent. Let's build the array *f(N)* starting with *N=1* :

N	*f(N)*	*result*
1	*1*	*you win*
2	*2*	*you win*
3	*3*	*you win*
4	*4*	*you win*
5	*5*	*you win*
6	×	*no optimal*

However we come across problem when *N=6*. No matter how many coins you remove, you are leaving either *f(1)*, *f(2)* , *f(3)* or … *f(5)* to your opponent who must win eventually. In other words there is no optimal strategy for *N=6* in which you must lose. When *N=7*, your optimal strategy is to leave a must-lose subgame to your opponent, which is *f(6)*.

N	*f(N)*	*result*
7	*1*	*leaving f(6) to opponent and you will win in next round*
8	*2*	*leaving f(6) to opponent and you will win in next round*
9	*3*	*leaving f(6) to opponent and you will win in next round*
10	*4*	*leaving f(6) to opponent and you will win in next round*
11	*5*	*leaving f(6) to opponent and you will win in next round*
12	×	*no optimal*

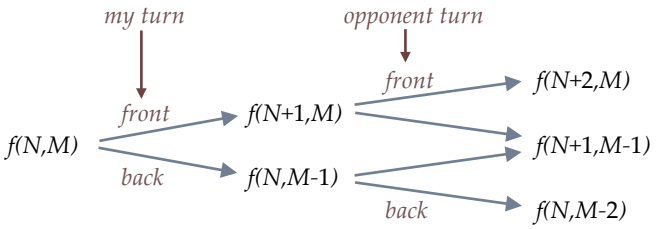
Therefore we have the optimal strategy for every *N*, and the array *f(N)* will repeat itself every *6* entries.

Two coin games – Game 2

Given a row of N coins with values $\{x_1, x_2, x_3, \ldots, x_N\}$, where N is even. We play a game against an opponent by alternating turns. In each turn, one player removes either the first coin or the last coin from the row and receives the value. Find the maximum value we can win if we move first, assuming that the opponent is as clever as we do.

Solution

As the game proceeds, we are facing a ***cropped portion*** the coin row, like the parenthesization problem. Likewisely we define $f(N,M)$ be the maximum value we can win from subgame $\{ x_N, x_{N+1}, x_{N+2}, \ldots, x_M \}$ where $N \le M$. Game theory kicks in (the game tree) :



- if we pick coin N, and
- if our opponent picks coin $N+1$ — then leave us with subgame $f(N+2,M)$
- if our opponent picks coin M — then leave us with subgame $f(N+1,M-1)$
- ⇒ thus opponent will pick so as to — $min(f(N+2,M), f(N+1,M-1))$

- if we pick coin M, and
- if our opponent picks coin N — then leave us with subgame $f(N+1,M-1)$
- if our opponent picks coin $M-1$ — then leave us with subgame $f(N,M-2)$
- ⇒ thus opponent will pick so as to — $min(f(N+1,M-1), f(N,M-2))$

Recursion is a max-min problem :

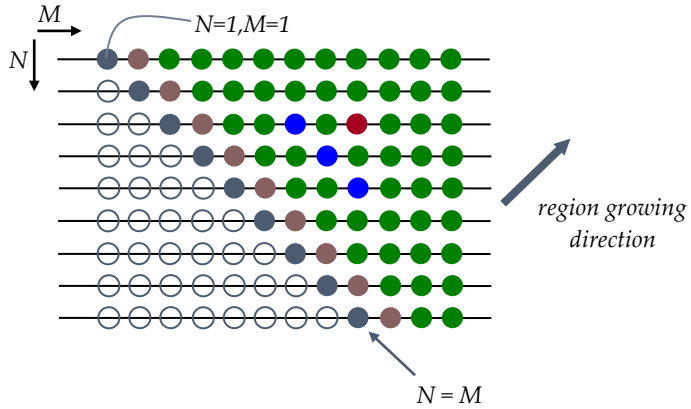
$$f(N,M) = \max\begin{pmatrix} x_N + \min(f(N+2,M), f(N+1,M-1)), \\ x_M + \min(f(N+1,M-1), f(N,M-2)) \end{pmatrix}$$

Boundary cases are :

$$f(N,M) = x_N \quad \text{if } M = N \quad \textit{diagonal nodes in subproblem-graph}$$

$$f(N,M) = \max(x_N, x_{N+1}) \quad \text{if } M = N+1 \quad \textit{sub-diagonal nodes in subproblem-graph}$$

As recursion depends on sub-problem 2 steps ahead, we have to initialize both diagonal and sub-diagonal for region growing.



Card coin from Deutsche Bank

Please refer to corresponding interview document.

Two persons traversal of cities

Given an ordered sequence of *M* cities in any dimensional space and a distance matrix *d(i,j)*, find an algorithm to divide the set into two such that when person *A* visits all cities in the first set in sequence while person *B* visits another set ***in sequence***, the total travel distance is minimised. This is a constrainted travelling salesman problem with *(1)* two salesmen and *(2)* predefined ordered cities.

Solution

Like longest common subsequence, we define *g(N,M)* as constrained version of original problem *f(M)* :

$g(N,M)$ = *minimum total distance if A ends at city N and B ends at city M, where N<M, N≠M and N,M ≥ 1*

then the original problem *f(M)* is solved by :

$$f(M) = \min_{N\in[1,M-1]} g(N,M)$$

- $g(N,M)$ is a symmetric matrix and we calculate *UR* corner only for sake of speed
- $g(N,M)$ is invalid on diagonal as A and B are not allowed to end at the same city
- $N < M$ means that all cities from *N+1* to *M* are visited by *B* only

Two recursions and one boundary case

off-diagonal (when M>N+1)

$$g(N,M) = \min[dist_A(x_A > \cdots > x_N) + dist_B(x_B > \cdots > x_{N+1} > x_{N+2} > \cdots > x_{M-1} > x_M)]$$

either one of x_A or x_B is x_1

$$= \min[dist_A(x_A > \cdots > x_N) + dist_B(x_B > \cdots > x_{N+1} > x_{N+2} > \cdots > x_{M-1})] + d(M-1,M)$$

$$= g(N,M-1) + d(M-1,M)$$

yet this eq is valid for N<M-1 only
for N=M-1, goto sub-diagonal case

sub-diagonal (when M=N+1)

$$g(N,M) = \min\begin{bmatrix} dist_A(x_A > \cdots > x_N) + dist_B(x_B > \cdots > x_{N-1} > x_{N+1}), \\ dist_A(x_A > \cdots > x_N) + dist_B(x_B > \cdots > x_{N-2} > x_{N+1}), \\ dist_A(x_A > \cdots > x_N) + dist_B(x_B > \cdots > x_{N-3} > x_{N+1}), \\ \ldots \\ dist_A(x_2 > x_3 > x_4 \cdots > x_N) + dist_B(x_B = x_1 > x_{N+1}), \\ dist_A(x_1 > x_2 > x_3 \cdots > x_N) + \underbrace{dist_B(x_B = x_{N+1})}_{0} \end{bmatrix}$$

by varying the 2nd last city for B

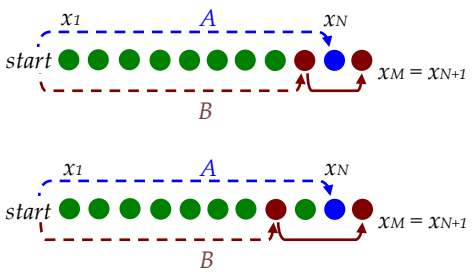
$$= \min\begin{bmatrix} dist_A(x_A > \cdots > x_N) + dist_B(x_B > \cdots > x_{N-1}) + d(N-1,N+1), \\ dist_A(x_A > \cdots > x_N) + dist_B(x_B > \cdots > x_{N-2}) + d(N-2,N+1), \\ dist_A(x_A > \cdots > x_N) + dist_B(x_B > \cdots > x_{N-3}) + d(N-3,N+1), \\ \ldots \\ dist_A(x_2 > x_3 > x_4 \cdots > x_N) + d(1,N+1), \\ dist_A(x_1 > x_2 > x_3 \cdots > x_N) \end{bmatrix}$$

$$= \min\left(\min_{n\in[1,N-1]}[g(n,N) + d(n,N+1)], \sum_{n=1}^{N-1} d(n,n+1)\right)$$

boundary case

$$g(1,2) = 0$$

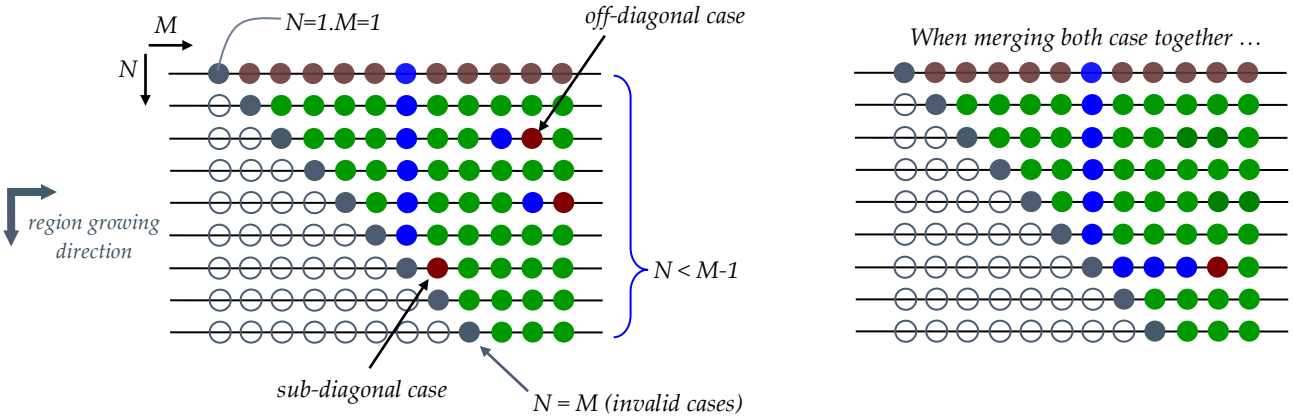
State-graph for case M = N+1



blue nodes and arrow = A
red nodes and arrow = B
green nodes = A/B

Subproblem-graph of g(N,M)

- this subproblem-graph is applicable for both *two-men traversal* and *bitonic tour*
- diagonal elements are invalid (as *A* and *B* cannot visit the same city in two-men traversal, or in open-bitonic tour)
- off-diagonal nodes depend on the previous node on *LHS*
- sub-diagonal nodes depend all nodes on previous column



Here are the implementations of two-men traversal involving 3 loops.

```
double two_men_tranversal(const std::vector<CITY>& cities,
                          std::function<double(const CITY&, const CITY&)>& dist)
{
    matrix f(cities.size(), cities.size());
    for(int n=0; n!=cities.size()-1; ++n)
    {
        for(int m=n+1; m!=cities.size(); ++m)
        {
            // *** case 1 *** //
            if (n < m-1) f(n,m) = f(n,m-1) + dist(cities[m-1], cities[m]);
            else
            {
                 // *** case 2.1 *** //
                 double min_value = 0;
                 for(int k=0; k!=n; ++k) min_value += dist(cities[k], cities[k+1]);

                 // *** case 2.2 *** //
                 for(int k=0; k!=n; ++k)
                 {
                      double temp = dist(cities[k], cities[m]);
                      if (min_value > f(k,n) + temp)
                          min_value = f(k,n) + temp;
                 }
                 f(n,m) = min_value;
            }
        }
    }

    // *** min of subproblems *** //
    double min_dist = 0;
    for(int n=0; n!=cities.size()-1;++n)
    {
        if (min_dist > f(n,cities.size()-1))
            min_dist = f(n,cities.size()-1);
    }
    return min_dist;
}
```

Bitonic tour

Given an ordered sequence of M cities in any dimensional space and a distance matrix $d(i,j)$, find the shortest bitonic tour. A bitonic tour is defined as the closed loop that passes through all points once, starting from the rightmost point x_N with decreasing index all the way to the leftmost point x_1, then return all the way back to x_N.

Solution

Let $f(M)$ be the shortest bitonic tour, it is consisted of :

- leftward trip $\{x_M \to ... \to x_1\}$
- rightward trip $\{x_1 \to ... \to x_M\}$
- as bitonic tour is a closed loop, $f(N)$ cannot be decomposed into sub-problem $f(N-1)$
- open-bitonic tour is thus introduced

Let $g(N,M)$ be the shortest open-bitonic tour, as one starts with x_M and ends at x_N where $N<M$, it is consisted of :

- leftward trip $\{x_M \to x_{M-1} \to \cdots \to x_{N+1} \to \cdots \to x_1\}$
- rightward trip $\{x_1 \to ... \to x_N\}$
- cities $x_{N+1}\ x_{N+2}\ x_{N+3}\ \ldots\ x_M$ should all be included in the leftward trip
- open-bitonic tour can be decomposed into sub-open-bitonic tour

$$g(N,M) = \min[dist_{leftward}(x_M \to x_{M-1} \to \cdots \to x_{N+1} \to \cdots \to x_1) + dist_{rightward}(x_1 \to \cdots \to x_N)]$$

$$f(M) = \min[dist_{leftward}(x_M \to \cdots \to x_1) + dist_{rightward}(x_1 \to \cdots \to x_M)]$$

$$= \min_{N \in [1, M-1]} g(N,M) + d(N,M)$$

This is a similar (but different) problem as two-men traversal problem :

- leftward tour is analgous to traveller B (as they covered x_M where $M>N$)
- rightward tour is analgous to traveller A (as they covered x_N)
- there are two extra segments in bitonic tour
- segment connecting first node of A and B
- segment connecting last node of A and B

Two recursions and one boundary case

off-diagonal (when M>N+1)

$$g(N,M) = \min[dist_{leftward}(x_M \to x_{M-1} \to \cdots \to x_{N+1} \to \cdots \to x_1) + dist_{rightward}(x_1 \to \cdots \to x_N)]$$

$$= d(M,M-1) + \min[dist_{leftward}(x_{M-1} \to \cdots \to x_{N+1} \to \cdots \to x_1) + dist_{rightward}(x_1 \to \cdots \to x_N)]$$

$$= g(N,M-1) + d(M-1,M)$$

sub-diagonal (when M=N+1)

$$g(N,M) = \min \begin{bmatrix} dist_{rightward}(x_{N+1} > x_{N-1} > \cdots > x_1) + dist_{leftward}(x_1 > \cdots > x_N), \\ dist_{rightward}(x_{N+1} > x_{N-2} > \cdots > x_1) + dist_{leftward}(x_1 > \cdots > x_N), \\ dist_{rightward}(x_{N+1} > x_{N-3} > \cdots > x_1) + dist_{leftward}(x_1 > \cdots > x_N), \\ \ldots \\ dist_{rightward}(x_{N+1} > x_2 > x_1) + dist_{leftward}(x_1 > \cdots > x_N), \\ dist_{rightward}(x_{N+1} > x_1) + dist_{leftward}(x_1 > \cdots > x_N) \end{bmatrix}$$

by varying the 2nd city in leftward trip

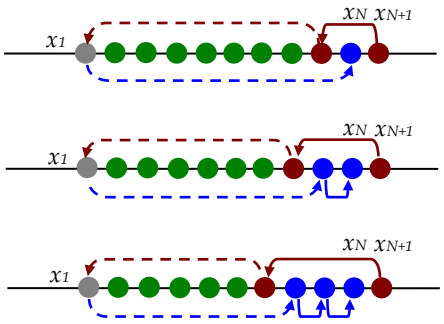
$$= \min \begin{bmatrix} d(N+1,N-1) + dist_{rightward}(x_{N-1} > \cdots > x_1) + dist_{leftward}(x_1 > \cdots > x_N), \\ d(N+1,N-2) + dist_{rightward}(x_{N-2} > \cdots > x_1) + dist_{leftward}(x_1 > \cdots > x_N), \\ d(N+1,N-3) + dist_{rightward}(x_{N-3} > \cdots > x_1) + dist_{leftward}(x_1 > \cdots > x_N), \\ \ldots \\ d(N+1,2) + dist_{rightward}(x_2 > x_1) + dist_{leftward}(x_1 > \cdots > x_N), \\ d(N+1,1) + dist_{leftward}(x_1 > \cdots > x_N) \end{bmatrix}$$

$$= \min_{n \in [1, N-1]} [g(n,N) + d(n,N+1)]$$

boundary case

$$g(1,2) = d(1,2)$$

State-graph for case M = N+1



blue nodes and arrow = leftward
red nodes and arrow = rightward
green nodes = leftward / rightward

Here are the implementations of bitonic tour problems involving 3 loops.

```
double bitonic_tour(const std::vector<CITY>& cities,
                    std::function<double(const CITY&, const CITY&)>& dist)
{
    matrix f(cities.size(), cities.size());
    for(int n=0; n!=cities.size()-1; ++n)
    {
        for(int m=n+1; m!=cities.size(); ++m)
        {
            // *** case 1 *** //
            if (n < m-1) f(n,m) = f(n,m-1) + dist(cities[m-1], cities[m]);
            else
            {
                 // *** case 2 *** //
                 double min_value = std::numeric_limits<double>::max();
                 for(int k=0; k!=n; ++k)
                 {
                      double temp = dist(cities[k], cities[m]);
                      if (min_value > f(k,n) + temp)
                          min_value = f(k,n) + temp;
                 }
                 f(n,m) = min_value;
            }
        }
    }

    // *** min of subproblems *** //
    double min_dist = 0;
    for(int n=0; n!=cities.size()-1;++n)
    {
        double temp = dist(cities[n], cities[cities.size()-1]);
        if (min_dist > f(n,cities.size()-1) + temp)
            min_dist = f(n,cities.size()-1) + temp;
    }
    return min_dist;
}
```

Piecewise linear regression

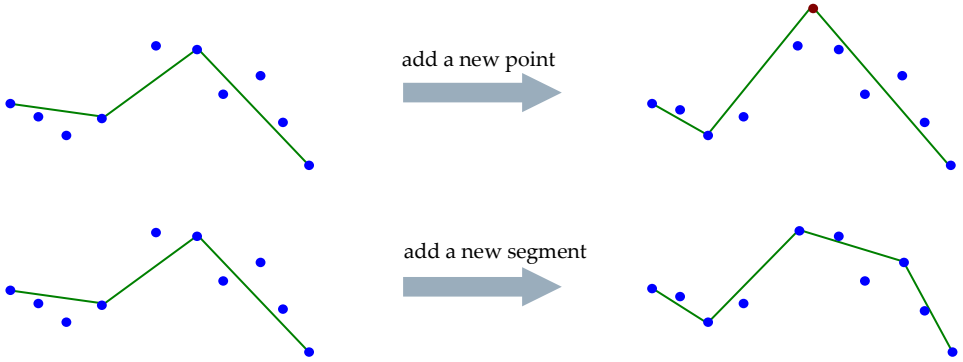
Given a sequence of N data points $\{(x_1,y_1), (x_2,y_2), (x_3,y_3), \ldots, (x_N,y_N)\}$ where $x_1 < x_2 < x_3 \ldots < x_N$, perform regression by piecewise linear function $f_M(x)$, having M segments, defined by $M+1$ control points, with minimum fitting error. The control points must be a subset of the dataset, with (x_1,y_1) and (x_N,y_N) as the first and the last control point. Fitting error of N data points using M segments is :

$$e(N,M) = \sum_{n=1}^{N}(f_M(x_n)-y_n)^2 \qquad \textit{where } N \geq M+1 \textit{ (case N=M+1 means each data point is a control point)}$$

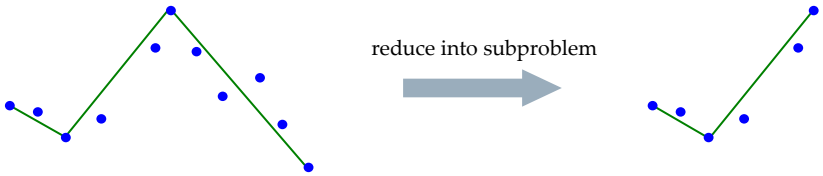
Solution

The original problem does not exhibit optimal substructure. We cannot find recursive relation :

- between $e(N,M)$ and $e(N-1,M)$ nor
- between $e(N,M)$ and $e(N,M-1)$ as shown in the following figure.



However, the optimal substructure happens when we consider the following subproblem :



Thus we have recursion of $e(N,M)$ depending on a bunch of $M-1$ segment models $e(N-1,M-1)$, $e(N-2,M-1)$, …, $e(M,M-1)$, with the last one denoting the case with all M data points being control points, hence $e(M,M-1) = 0$.

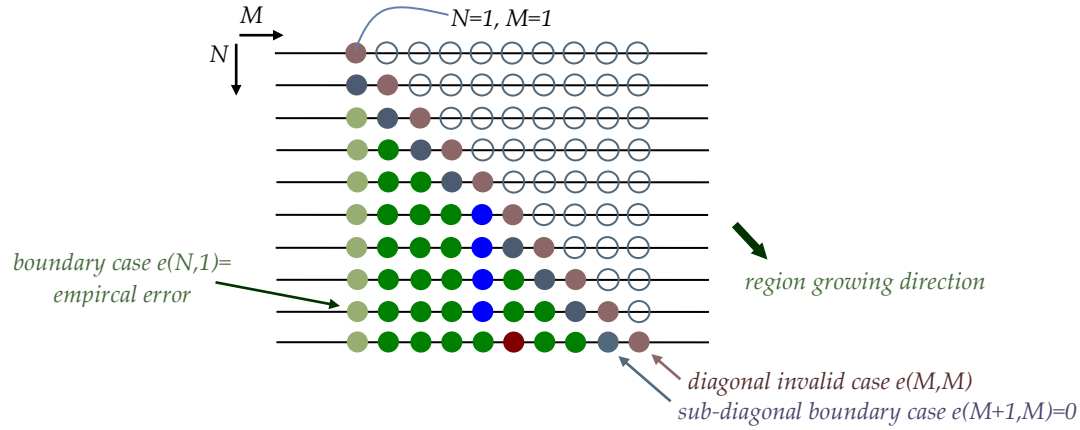
$$e(N,M) = \min_{n\in[M,N-1]}(e(n,M-1)+\sum_{i=n}^{N}(g(x_i \mid x_n,y_n,x_N,y_N)-y_i)^2) \qquad \textit{recursion}$$

$$e(M+1,M) = 0 \qquad \textit{boundary case (sub-diagonal)}$$

$$e(N,1) = \sum_{i=n}^{N}(g(x_i \mid x_1,y_1,x_N,y_N)-y_i)^2 \qquad \textit{boundary case (first column)}$$

$$s.t. \quad g(x \mid x_n,y_n,x_N,y_N) = \frac{y_N-y_n}{x_N-x_n}\times(x-x_n)+y_n \qquad \textit{the line joining } (x_n,y_n) \textit{ and } (x_N,y_N)$$

Subproblem graph for *e(N,M)* is :



Here are the implementation.

```
template<typename ITER> double calculate_error(ITER begin, ITER end) // constraint : begin < end-1
{
    ITER ptr0 = begin;
    ITER ptr1 = end-1;
    double sum_error2 = 0;
    for(ITER i = ptr0+1; iter!=ptr1; ++iter)
    {
        double error = (i->y-ptr0.y) - (i->x-ptr0.x)/(ptr1.x-ptr0.x)*(ptr1.y-ptr0.y);
        sum_error2 += error * error;
    }
    return sum_error2;
}

double piecewise_error(const std::vector<point>& points, int num_segments)
{
    mat f(points.size(), num_segments); // diagonal is useless, UR is useless

    // boundary condition
    for(int n=1; n!=points.size(); ++n)
    {
        f(n,0) = calculate_error(points.front(), points.back(), points.begin()+1, points.end()-1);
    }

    // region growing : redundant calculation for loop m == num_segment-1 ...
    for(int m=1; m!=num_segments; ++m)
    {
        //     num of segments = m+1
        // min num of points   = m+1+1
        // min n               = m+1
        for(int n=m+1; n!=points.size(); ++n)
        {
            f(n,m) = std::numeric_limits<double>::max();
            for(int k=m; k!=n; ++k)
            {
                double error = f(k,m-1) + calculate_error(points.begin()+k, points.end());
                if (error < f(n,m)) f(n,m) = error;
            }
        }
    }
    return f(points.size()-1, num_segments-1);
}
```

Box stacking

Given N boxes, having sides $x_{n,1}$, $x_{n,2}$ and $x_{n,3}$ $\forall n \in [1,N]$. Find is the maximum height of stack of boxes, so that :

- each box should be used once or never
- box i should be placed above box j for $i < j$
- base of any box is *strictly smaller than* base of box lying below
- orientation of boxes can be changed

Bin packing

Given N_1 objects with size s_1, N_2 objects with size s_2 and N_3 objects with size s_3, find the minimum number of boxes required to pack all the objects, assuming the capacity of box is C, such that $s_1 \leq s_2 \leq s_3 \leq C$.

Solution to box stacking

Lets define $f(N,b_1,b_2)$ as maximum height of box-stack with a set of N boxes, with base constraint $b_1 \times b_2$.

$$max\text{-}stack = f(N,\infty,\infty)$$

$$f(N,b_1,b_2) = \max\begin{bmatrix} x_{N,1} + f(N-1,x_{N,2},x_{N,3}) & if & (x_{N,2},x_{N,3}) < (b_1,b_2) \\ x_{N,2} + f(N-1,x_{N,3},x_{N,1}) & if & (x_{N,3},x_{N,1}) < (b_1,b_2) \\ x_{N,3} + f(N-1,x_{N,1},x_{N,2}) & if & (x_{N,1},x_{N,2}) < (b_1,b_2) \\ f(N-1,b_1,b_2) & & \end{bmatrix}$$

$$f(0,b_1,b_2) = 0$$

```
struct box { double x; double y; double z; };
bool fit(double x, double y, double limit0, double limit1)
{
    return (x <= limit0 && y <= limit1) || (x <= limit1 && y <= limit0);
}

double box_stack(const std::vector<box>::iterator begin,
                 const std::vector<box>::iterator end,
                 double limit0 = std::numeric_limits<double>::max(),
                 double limit1 = std::numeric_limits<double>::max())
{
    const box b = *begin;
    double temp = max_stack(begin+1, end, limit0, limit1);
    if (fit(b.x, b.y, limit0, limit1))
    {
        if (temp < b.z + max_stack(begin+1, end, b.x, b.y))
            temp = b.z + max_stack(begin+1, end, b.x, b.y);
    }
    if (fit(b.y, b.z, limit0, limit1))
    {
        if (temp < b.x + max_stack(begin+1, end, b.y, b.z))
            temp = b.x + max_stack(begin+1, end, b.y, b.z);
    }
    if (fit(b.z, b.x, limit0, limit1))
    {
        if (temp < b.y + max_stack(begin+1, end, b.z, b.x))
            temp = b.y + max_stack(begin+1, end, b.z, b.x);
    }
    return temp;
}
```

Solution to bin packing

This problem is like parenthesization problem in *3D* :

- picking one point as parent node
- apply recursion on partitions to generate subtree

Suppose $f(N_1,N_2,N_3)$ be the minimum number of boxes needed, by breaking it down into two subproblems at cutoff point (n_1,n_2,n_3) we have two subproblems as $f(n_1,n_2,n_3)$ and $f(N_1\text{-}n_1,N_2\text{-}n_2,N_3\text{-}n_3)$. By solving the two subproblems, merging the results, repeating the process by iterating through all possible cutoff points, we can find the optimum.

$$f(N_1,N_2,N_3) = \begin{bmatrix} \min\limits_{\substack{n_1\in[0,N_1] \\ n_2\in[0,N_2] \\ n_3\in[0,N_3] \\ n_1n_2n_3\neq 0}} (f(N_1-n_1,N_2-n_2,N_3-n_3)+f(n_1,n_2,n_3)) & if & N_1s_1+N_2s_2+N_3s_3 > C \\ 1 & if & N_1s_1+N_2s_2+N_3s_3 \le C \end{bmatrix}$$

```
int bin_pack(int num_x, int num_y, int size_x, int size_y, int size_box)
{
    if (num_x*size_x + num_y*size_y <= size_box) return 1;
    int min_count = num_x + num_y;
    for(int n=0; n<=num_y; ++n)
    {
        for(int m=0; m<=num_x; ++m)
        {
            if (n==0     && m==0)     continue;
            if (n==num_y && m==num_x) continue;
            int temp =  min_box_needed(m, n, size_x, size_y, size_box) +
                        min_box_needed(num_x-m, num_y-n, size_x, size_y, size_box);
            if (min_count > temp)
                min_count = temp;
        }
    }
    return min_count;
}
```

Viterbi Algorithm

This is a special case of Dijkstra algorithm. When the state-graph is a multipartite graph, or network which is defined as specialized graph having ***no edge between two non-adjacent layers***, then region growing can be done by iterating through each layer, and each node in a layer, without use of priority queue. This is known as Viterbi algorithm.

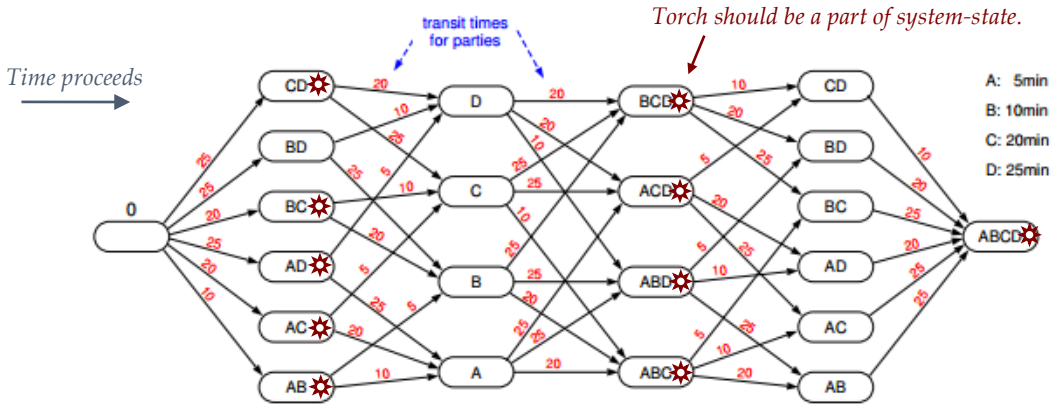
- Dijkstra algorithm is region growing in directed acyclic graph using priority queue
- Viterbi algorithm is region growing in multi-partite graph without priority queue

Suppose *4* persons *A*, *B*, *C* and *D* want to traverse a suspension bridge at night. Here are the rules :

- the bridge can carry no more than two persons at a time,
- the four persons take *5*, *10*, *20* and *25* minutes respectively to traverse,
- each traversal requires a torch which lasts no longer than *60* minutes.

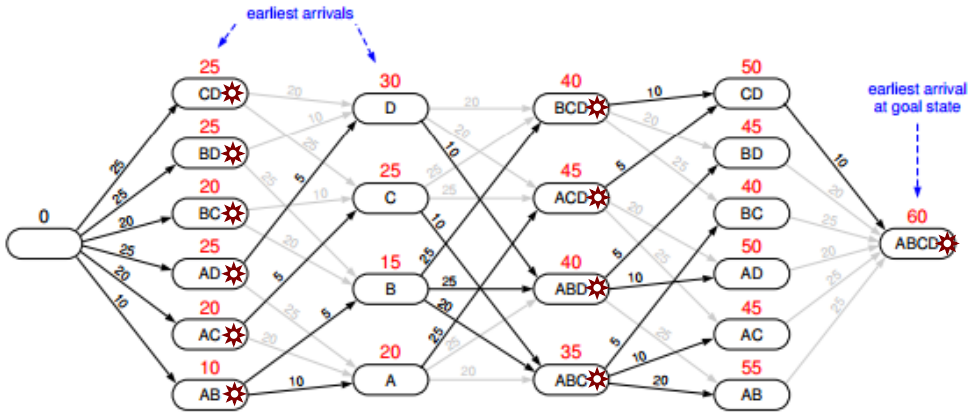
Step 1 – Setup state-graph

A state is defined as the set of persons (including the torch) on the opposite side of the bridge. Each edge represents a possible state transition. Some state transitions are not allowed, for example it is impossible to transit from state *0* to state *ABC*.



Step 2 – Growing the graph

Propagate the shortest time of reaching the states in next partite.



We can trace out the shortest path and the total time needed using backward propagation.

Remarks

- This is a state-graph. If we consider the node with torch only, it becomes a subproblem-graph.
- Any problem that is transformable into multipartite graph by introducing time element can be solved using Viterbi.

Retrospective trader

Given N ticks of stock price $\{x_1, x_2, x_3, \ldots, x_N\}$, find the optimum position-path that maximizes profit, under constraints :

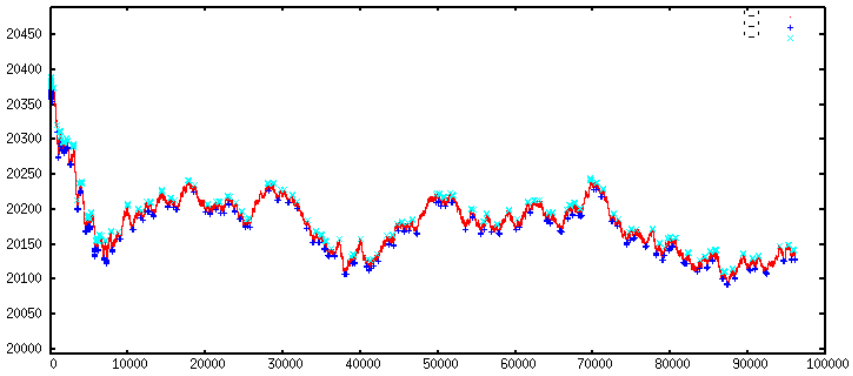
- maximum lots per trade
- maximum position
- short sell permission
- zero position by day-end
- transaction cost consideration

Solution

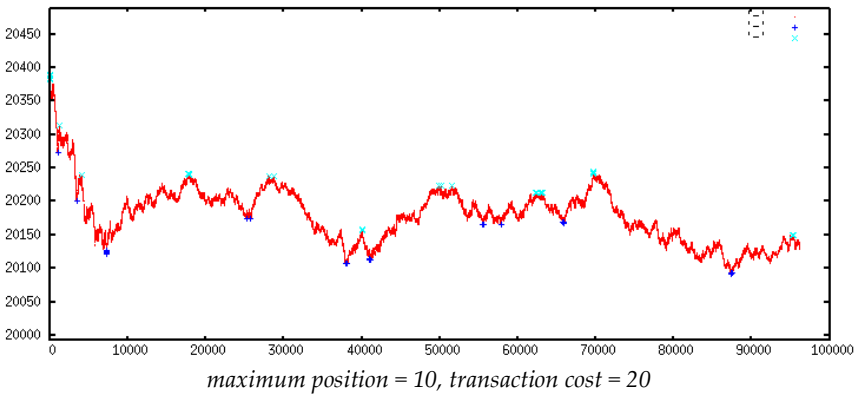
This problem can be addressed as a state graph with $N+1$ layers, layer 0 denotes initial state, that is zero position, while the n^{th} layer denotes all possible states after reception of the n^{th} ticks, i.e. states after n iterations. Possible states are the number of shares that we are allowed to have at each iteration, and each edge represents a transaction made, therefore it is a change in position. The value on each vertex is the net cash at that state, all transaction costs should be deducted from the net cash. This is a multipartite graph, with no edge between any two non-adjacent layers. Dynamic programming is thus a $2D$ scan :

```
for(int n=0; n!=N; ++n) // n = iteration
{
    for(int m0=-M; m0<=M; ++m0) // m0 = number of shares at n, where position limit = [-M,+M]
    {
        for(int m1=-M; m1<=M; ++m1) // m1 = number of shares at n+1, where position limit = [-M,+M]
        {
            else if (within_trading_limit(m0,m1))
            {
                auto cashflow = tick(n).price()*(m0-m1) - tran_cost; // -ve cashflow for buy, +ve cashflow for sell
                node(n+1,m1).cash = std::max(node(n+1,m1).cash, node(n,m0).cash + cashflow);
            }
        }
    }
} // This is a very naïve implementation, please add link to ancestor, number of trade, sell all by dayend.
```

Finally we can extract the optimum position path by backward induction. Here are test results for *HSIF* with blue dots representing buy signals and cyan dots representing sell signals. It is liked running a support vector regression on the price time series, with half tube width equals to transaction cost. Buy signals form a set of negative support vectors, whereas sell signals form a set of positive support vectors.



maximum position = 10, transaction cost = 5



maximum position = 10, transaction cost = 20

Implementation of retrospective trader. Useful pattern with two `std::unordered_map`, no need to build a graph.

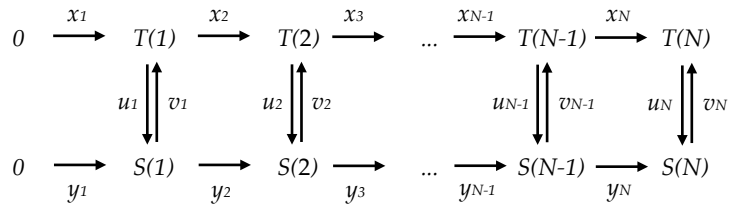
```
double retrospective_trader(const std::vector<double>& stock, int trade_limit)
{
    std::unordered_map<int, cash> f;
    f[0] = 0;

    for(const auto& St : stock)
    {
        std::unordered_map<int, cash> g;
        for(auto& x:f)
        {
            for(int n=-trade_limit; n<=trade_limit; ++n)
            {
                auto pos  = x.first + n;
                auto cash = x.second - n * St;
                if (position_check(pos))
                {
                    auto iter = g.find(pos);
                    if (iter == g.end()) g[pos] = cash;
                    else if (iter->second < cash)
                             iter->second = cash;
                }
            }
        }
        // *** MOVE *** //
        f = std::move(g);
    }
    return f[0];
}
```

Assembly line pairs

Suppose there are 2 production lines producing the same product, each line has N steps, with processing time $x_1, x_2, x_3, \ldots, x_N$ and $y_1, y_2, y_3, \ldots, y_N$ respectively. Right after completing step n we can transfer intermediately in between the two lines with transfer time u_n and v_n respectively for both directions, where n=1,2,…,N. What is the minimum time for completing all N steps?

Solution

Suppose $T(N)$ and $S(N)$ be the time needed to reach a particular state in production line *1* and *2* respectively, the state-graph is :



This is a modified version Viterbi algorithm which has links within the same layer. The recursion is :

$$T(N) = \min(T(N-1)+x_N, S(N-1)+y_N+v_N)$$
$$S(N) = \min(S(N-1)+y_N, T(N-1)+x_N+u_N)$$

Change making problem

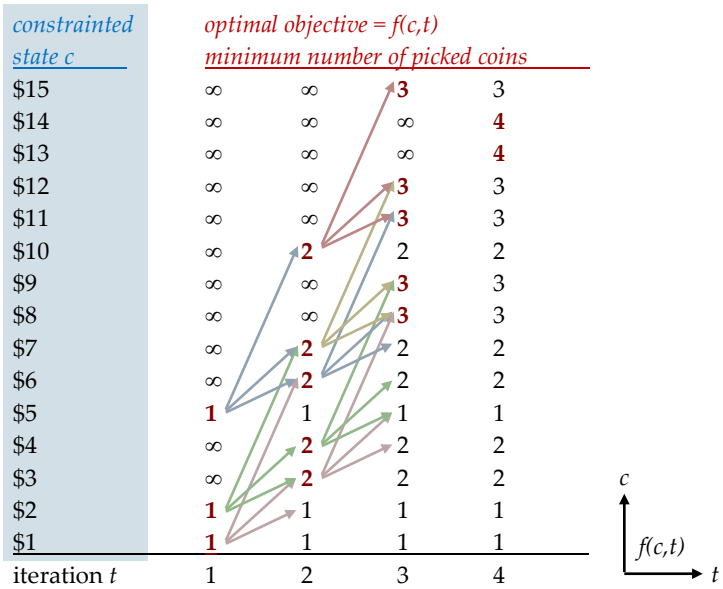
Given N types of coins with integer values $\{x_1, x_2, x_3, \ldots, x_N\}$, such that $x_1 = 1 < x_2 < x_3 \ldots < x_N$, there are infinite many number for each of them. Design an algorithm for finding the smallest number of coins, which sum to a given integer value C exactly.

Solution

Define $f(c,t)$ as the minimum number of coins needed to give a sum c, if we are allowed to pick t coins at maximum :

- vertex state c is the sum after t iterations ***(like budget constrain in microeconomics)***
- vertex value $f(c,t)$ is the ***minimum number of coins*** needed
- each vertex has N edges connecting to next iteration
- each edge introduces ***extra cost of 1 for addition coin***

Suppose we have coin *$1*, *$2* and *$5* only, the multipartite graph for $f(c,t)$ can be illustrated as :



All Viterbi problems can be solved by either :
(1) iteration starting from LL corner
(2) recursion starting from UR corner
Both methods apply the recursion formula.

- Updated values in each step is highlighed in **red**.
- Keep growing in time-domain until C is reached.
- No need to keep a matrix $f(c,t)$, what we need is to keep two vectors and apply move semantics.
- Recursion formula (for both iterative or recursive approach) in each step is :

$$f(c,t) = \min_{k} \sum_{n\in[1,N]} k_n 1 \quad such\ that \quad \sum_{n\in[1,N]} k_n x_n = c$$

where k_n coins with face value x_n are picked

$$= \begin{cases} \min\limits_{n\in[1,N],c\geq x_n} (f(c-x_n,t-1)+1) & if \quad c>0 \\ 0 & if \quad c=0 \end{cases}$$

```
int min_num_coin(const std::vector<int>& coins, int target)
{
    std::unordered_map<int, int> f; f[0] = 0;
    while(true)
    {
        std::unordered_map<int, int> g;
        for(auto& x:f)
        {
            for(int n=0; n!=coins.size(); ++n)
            {
                auto value = x.first + coin[n];
                auto count = x.second + 1; // <--- change this line for knapsack
                auto iter = g.find(value);
                if (iter == g.end()) g[value] = count;
                else if (iter->second > count) iter->second = count;
            }
        }

        // *** EXIT when DONE *** // <--- quiting condition for knapsack will be different
        auto iter = g.find(target);
        if (iter != g.end()) return iter->second;

        // *** MOVE *** //
        f = std::move(g);
    }
}
```

Knapsack problem

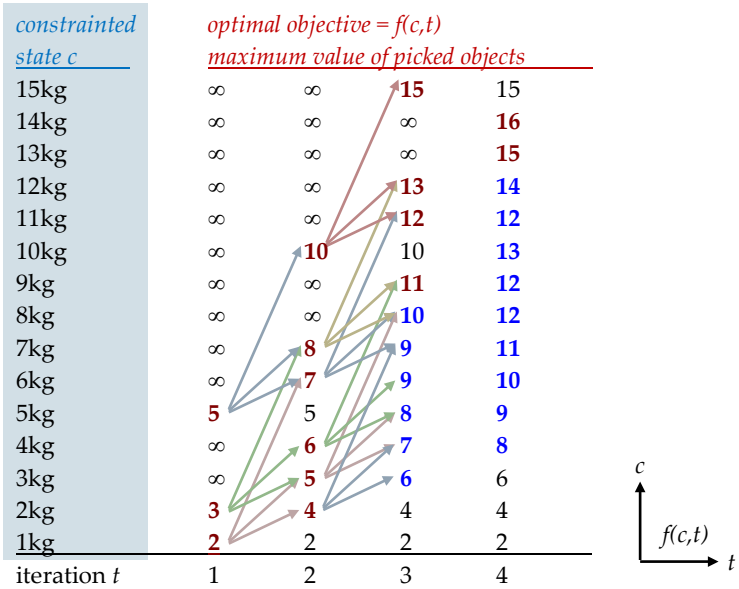
Given N types of object with weights $\{x_1, x_2, x_3, \ldots, x_N\}$ and values $\{y_1, y_2, y_3, \ldots, y_N\}$, there are infinite many number for each of them. Design an algorithm for finding maximum value of objects packed inside a knapsack, which has a maximum load of C.

Solution

Define $f(c,t)$ as the maximum value of objects with total weight c, if we are allowed to pick t objects at maximum :

- vertex state c is the total weight after t iterations
- vertex value $f(c,t)$ is the ***maximum value of objects***
- each vertex has N edges connecting to next iteration
- each edge introduces ***extra value for addition object***

Suppose we have objects of *1kg*, *2kg* and *5kg* only, with value *2*, *3* and *5*, the multipartite graph $f(c,t)$ can be illustrated as :



- Re-updated values in each step is highlighed in **blue** *(there is no blue update in change making problem)*.
- Keep growing in iteration-domain until all states go beyond C.
- No need to keep a matrix $f(c,t)$, what we need is to keep two vectors and apply move semantics.
- Recursion formula (for both iterative or recursive approach) in each step is :

$$f(c,t) = \max_{k} \sum_{n\in[1,N]} k_n y_n \quad \text{such that} \quad \sum_{n\in[1,N]} k_n x_n = c \qquad \text{where } k_n \text{ coins with face value } x_n \text{ are picked}$$

$$= \begin{cases} \max\limits_{n\in[1,N],c\geq x_n} (f(c-x_n, i-1)+y_n) & \text{if} \quad c > \min(x_1,x_2,x_3,\ldots,x_N) \\ 0 & \text{if} \quad c < \min(x_1,x_2,x_3,\ldots,x_N) \end{cases}$$

Minimization vs Maximization

Unlike change-making problem which has `while(true)` loop being broken when target C is touched for the first time, however in this question, we have to loop until a time instance when all possible states are all above the maximum load C. The main reason for the difference is that the fomer question is about minimization (just like Dijkstra shortest path), but this question is about maximization.

Job scheduling

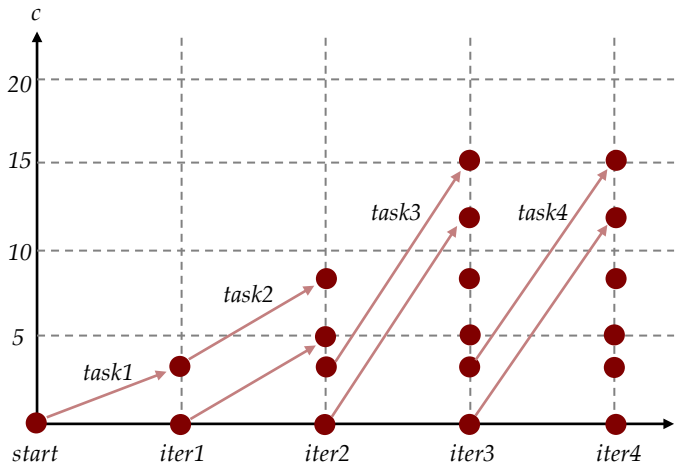
Given one machine and N jobs, each job requires processing time x_n and generates profit y_n if it can be done within deadline C_n. The machine can only process one job at a time, each job is atomic. Find maximum profit, assuming $x_n \le C_n \ \forall n \in [1,N]$ and $C_1 \le C_2 \le \dots$

Solution

Define $f(c, X)$ as the maximum profit of tasks given a whole set of task $X = \{x_1, x_2, x_3, \dots, x_N\}$:

- vertex state c is total man-day offered
- vertex value $f(c, X)$ is the ***maximum profit of tasks***
- each vertex has N-n edges connecting to next iteration, since tasks are done in order
- each edge introduces ***extra profit for addition task***

Suppose we have 4 tasks as shown in the table, the inductive multipartite graph $f(c, X)$ can be illustrated as :



Task	effort	deadline	profit
1	3	5	p1
2	5	10	p2
3	12	15	p3
4	12	16	p4

← *Update the maximum profit at each vertex.*

Recursion formula (for both iterative or recursive approach) in each step is :

$$f(c,X) = \begin{cases} f(c, X \setminus x_1) & \text{if} \quad c < x_1 \\ \max(f(c, X \setminus x_1), f(c - x_1, X \setminus x_1) + y_1) & \text{if} \quad c \ge x_1 \end{cases}$$

consider first job as its deadline is closest

Equal partition sum

Given set of integers $X = \{x_1, x_2, \dots, x_N\}$, partition the set into two so that absolute difference between two subset sums is minimised.

Solution

Define $f(c, X)$ as the boolean indicating whether the sum c can be reachable by picking integers from set X. In order to avoid reusing the same integer, we have to store all used integers for each path. Suppose we have integers [1, 2, 5], multipartite graph :

state c	reachable (true or false)		
8	-	-	1,2,5
7	∞	2,5	2,5
6	∞	1,5	1,5
5	5	5	5
4	-	-	-
3	-	1,2	1,2
2	2	2	2
1	1	1	1
iteration t	1	2	3

Recursion formula (for both iterative or recursive approach) in each step is :

$$f(c,X) = \begin{cases} f(c, X \setminus x_1) \cap f(c - x_1, X \setminus x_1) & \text{if} \quad x \ne x_1 \\ true & \text{if} \quad x = x_1 \end{cases}$$

where X is the input set of integers

If $f(floor(\frac{1}{2}\sum_{n=1}^{N} x_n), X) = true$ *then equal-partite exists.*

Comparison among coin change / knapsack / job scheduling / partition

The seven elements for various problems.

	making change problem			*knapsack problem*			*job scheduling*		
1	*coin value*	=	$\{x_1, x_2, ..., x_N\}$	*object weight*	=	$\{x_1, x_2, ..., x_N\}$	*task man-day*	=	$\{x_1, x_2, ..., x_N\}$
2	*coin count*	=	1	*object value*	=	$\{y_1, y_2, ..., y_N\}$	*task proft*	=	$\{y_1, y_2, ..., y_N\}$
3	*coin num*	=	$\{k_1, k_2, ..., k_N\}, k_n \in \aleph$	*object num*	=	$\{k_1, k_2, ..., k_N\}, k_n \in \aleph$	*object num*	=	$\{k_1, k_2, ..., k_N\}, k_n \in [0,1]$
4	*time*	=	i	*time*	=	i	*time*	=	$X \setminus x_1$ *(reducing set)*
5	*objective*	=	$\min \sum_{n=1}^{N} k_n$	*objective*	=	$\max \sum_{n=1}^{N} k_n y_n$	*objective*	=	$\max \sum_{n=1}^{N} k_n y_n$
6	*constrain*	=	$\sum_{n=1}^{N} k_n x_n = C$	*constrain*	=	$\sum_{n=1}^{N} k_n x_n \le C$	*constrain*	=	$\sum_{n=1}^{1} k_n x_n \le C_1$ $\sum_{n=1}^{2} k_n x_n \le C_2$ … $\sum_{n=1}^{N} k_n x_n \le C_N$
7	*recursion*								

$f(c,t)$	=	$\min_{n \in [1,N]} f(c - x_n, t-1) + 1$	*for making change*
$f(c,t)$	=	$\max_{n \in [1,N]} f(c - x_n, t-1) + y_n$	*for knapsack problem*
$f(c,X)$	=	$\max[f(c, X \setminus x_1), f(c - x_1, X \setminus x_1) + y_1]$	*for job scheduling*
$f(c,X)$	=	$f(c, X \setminus x_1) \cap f(c - x_1, X \setminus x_1)$	*for partition*