CS 260
Design and Analysis of Algorithms
4. Dynamic Programming

Mikhail Moshkov

Computer, Electrical and Mathematical Sciences & Engineering Division
King Abdullah University of Science and Technology

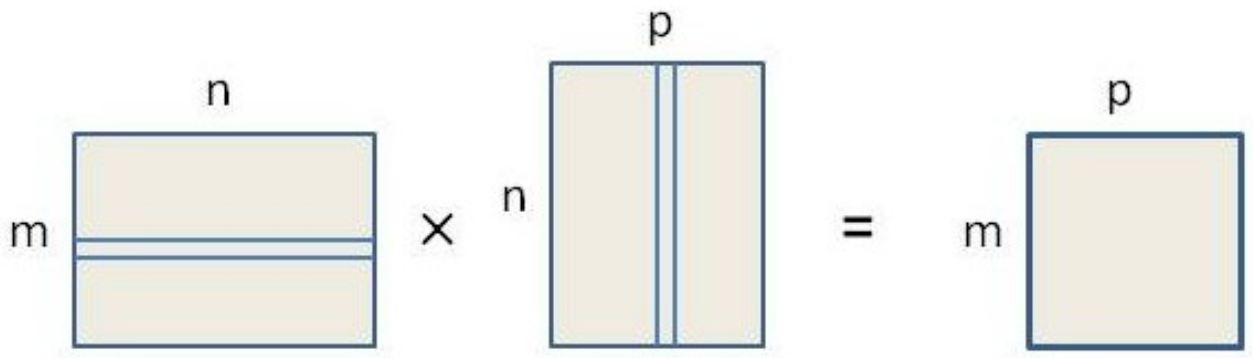
Dynamic Programming

The idea of dynamic programming is the following. For a given problem, we define the notion of a subproblem and an ordering of subproblems from "smallest" to "largest".

If (i) the number of subproblems is polynomial, and (ii) the solution of a subproblem can be easily (in polynomial time) computed from the solution of smaller subproblems then we can design a polynomial algorithm for the initial problem.

Chain Matrix Multiplication

We will consider the usual way for matrix multiplication and we will study only the number of element multiplications. Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications.



Matrix multiplication is *associative*: $A \times (B \times C) = (A \times B) \times C$.

Chain Matrix Multiplication

Let us consider matrixes A, B, C with dimensions $n \times 1$, $1 \times n$, and $n \times n$, respectively.

The dimension of AB is equal to $n \times n$. So the parenthesization $(AB)C$ requires $n^2 + n^3$ multiplications.

The dimension of BC is $1 \times n$. Therefore the parenthesization $A(BC)$ requires only $n^2 + n^2$ multiplications. This number is essentially less than for the first case.

Chain Matrix Multiplication

Let we have n matrices $A_1, A_2, \ldots, A_n$ with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$, respectively.

We should find the minimal cost of multiplying $A_1 \times A_2 \times \ldots \times A_n$ – the minimum number of multiplications of elements among all possible parenthesizations. Also we would like to find an optimal parenthesization.

The brute-force approach is a bad way: for three matrixes we have two parenthesizations, so for $3n$ matrixes we have at least 2^n possible parenthesizations.

Chain Matrix Multiplication

Let us consider the following subproblems B_{ij}, $1 \leq i \leq j \leq n$, for the initial problem: to find the minimum cost $C(i, j)$ of multiplying of $A_i \times A_{i+1} \times \ldots \times A_j$ and an optimal parenthesization.

The size of this subproblem is the number $j - i$. The smallest subproblem is when $i = j$, in which case there is nothing to multiply, so $C(i, i) = 0$.

Chain Matrix Multiplication

Let $j > i$. Let us show that

$$C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}.$$

If the last operation of matrix multiplication divides the product $A_i \times A_{i+1} \times \ldots \times A_j$ into two subproducts $(A_i \times \ldots \times A_k)(A_{k+1} \times \ldots \times A_j)$ then to obtain the minimal number of element multiplications we should use the minimum number of element multiplications in the both subproducts and $m_{i-1} \cdot m_k \cdot m_j$ element multiplications to multiply the two subproducts.

To obtain the minimal cost we should choose k for which the expression in braces has minimum value.

Chain Matrix Multiplication

The number of subproblems is $O(n^2)$.

If we know $C(i,k)$ and $C(k+1,j)$, $i \leq k < j$, than to compute $C(i,j)$ it is enough to make $O(n)$ operations of element multiplication, addition and comparison.

So the whole number of such operations to compute $C(1,n)$ is $O(n^3)$.

If during the computation of $C(i,j)$ we fix k for which $C(i,j)$ has minimal value, then we will be able to restore an optimal parenthesization.

Chain Matrix Multiplication

Example 4.1. Chain Matrix Multiplication

A_1	A_2	A_3	A_4	$A_1 \times A_2 \times A_3 \times A_4$
2×3	3×4	4×5	5×2	
$m_0 \times m_1$	$m_1 \times m_2$	$m_2 \times m_3$	$m_3 \times m_4$	$B_{ij}, 1 \leq i \leq j \leq 4$

$$C(i,j) = \min_{i \leq k < j}\{C(i,k) + C(k+1,j) + m_{i-1} m_k m_j\}$$

$$C(1,1) = 0,\ C(2,2) = 0,\ C(3,3) = 0,\ C(4,4) = 0$$

$$C(1,2)_{k=1} = \left\{ \begin{array}{c} C(1,1) + C(2,2) + m_0 m_1 m_2 \\ 0 + 0 + 2 \cdot 3 \cdot 4 \end{array} \right\} = 24,$$

$$C(2,3)_{k=2} = \left\{ \begin{array}{c} C(2,2) + C(3,3) + m_1 m_2 m_3 \\ 0 + 0 + 3 \cdot 4 \cdot 5 \end{array} \right\} = 60,$$

$$C(3,4)_{k=3} = \left\{ \begin{array}{c} C(3,3) + C(4,4) + m_2 m_3 m_4 \\ 0 + 0 + 4 \cdot 5 \cdot 2 \end{array} \right\} = 40$$

Chain matrix multiplication

Example 4.1 (continuation).

$C(i,j) = \min_{i \le k < j}\{C(i,k) + C(k+1,j) + m_{i-1} m_k m_j\}$

$m_0 = 2,\ m_1 = 3,\ m_2 = 4,\ m_3 = 5,\ m_4 = 2,$
$C(1,1) = C(2,2) = C(3,3) = C(4,4) = 0,\ C(1,2) = 24,$
$C(2,3) = 60,\ C(3,4) = 40$

$$C(2,4)_{k=2,3} = \min \left\{ \begin{array}{ll} C(2,2) + C(3,4) + m_1 m_2 m_4, & C(2,3) + C(4,4) + m_1 m_3 m_4 \\ 0 + 40 + 3 \cdot 4 \cdot 2 = 64, & 60 + 0 + 3 \cdot 5 \cdot 2 = 90 \end{array} \right\} = 64$$

$$C(1,3)_{k=1,2} = \min \left\{ \begin{array}{ll} C(1,1) + C(2,3) + m_0 m_1 m_3, & C(1,2) + C(3,3) + m_0 m_2 m_3 \\ 0 + 60 + 2 \cdot 3 \cdot 5 = 90, & 24 + 0 + 2 \cdot 4 \cdot 5 = 64 \end{array} \right\} = 64$$

Chain Matrix Multiplication

Example 4.1 (continuation).

$C(i,j) = \min_{i \le k < j}\{C(i,k) + C(k+1,j) + m_{i-1} m_k m_j\}$

$m_0 = 2,\ m_1 = 3,\ m_2 = 4,\ m_3 = 5,\ m_4 = 2,$
$C(1,1) = C(2,2) = C(3,3) = C(4,4) = 0,\ C(1,2) = 24,$
$C(2,3) = 60,\ C(3,4) = 40,\ C(2,4) = 64,\ C(1,3) = 64,$

$$C(1,4)_{k=1,2,3} = \min \left\{ \begin{array}{ll} C(1,1) + C(2,4) + m_0 m_1 m_4, & C(1,2) + C(3,4) + m_0 m_2 m_4, \\ 0 + 64 + 2 \cdot 3 \cdot 2 = 76, & 24 + 40 + 2 \cdot 4 \cdot 2 = 80, \\ C(1,3) + C(4,4) + m_0 m_3 m_4 & \\ 64 + 0 + 2 \cdot 5 \cdot 2 = 84 & \end{array} \right\} = 76$$

Optimal parenthesization: $A_1 \times (A_2 \times (A_3 \times A_4))$

Shortest Paths

In this section, we consider Floyd-Warshall algorithm which is a dynamic programming algorithm.

Let G be a complete directed graph with n nodes $v_1, \ldots, v_n$ in which each direct edge (v_i, v_j) is labeled with d_{ij} where d_{ij} is a real number or $d_{ij} = +\infty$. The number d_{ij} is interpreted as the length of the directed edge (v_i, v_j). We will assume that $d_{ii} = 0$, $i = 1, \ldots, n$.

The length of a directed path from v_i to v_j is equal to the sum of lengths of edges in this path (the length of a path is equal to $+\infty$ if the length of at least one edge from the path is equal to $+\infty$).

Shortest Paths

It is possible that $d_{ij} < 0$ but we assume that there are no directed cycles with negative length (negative cycles).

The minimum distance d^*_{ij} from v_i to v_j is equal to the minimum length of a directed path from v_i to v_j.

For a given $n \times n$ matrix $D = \|d_{ij}\|$ of lengths of edges we should construct the matrix $D^* = \left\|d^*_{ij}\right\|$ of minimal distances.

Shortest Paths

For any $i, j \in \{1, \ldots, n\}$ and $k \in \{0, 1, \ldots, n\}$ let us consider the following subproblem: to compute $d_{ij}^{(k)}$ which is the length of the shortest path from v_i to v_j in which only nodes $v_1, \ldots, v_k$ can be used as intermediate nodes. If $k = 0$ then $d_{ij}^{(0)} = d_{ij}$.

Let $D^{(k)} = \left\| d_{ij}^{(k)} \right\|$. Then $D^{(0)} = D$ and $D^{(n)} = D^*$. We will sequentially compute $D^{(1)}, \ldots, D^{(n)}$.

Shortest Paths

Let us prove that, for every i, j and $k > 0$,

$$d_{ij}^{(k)} = \min \left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\}.$$

All directed paths from v_i to v_j that use only $v_1, \ldots, v_k$ as intermediate nodes can be divided into two sets A and B – which do not pass through v_k and which pass through v_k.

The minimum length of a path from A is equal to $d_{ij}^{(k-1)}$.

Shortest Paths

Since there are no negative cycles in G, there exists a shortest path τ from B which passes through v_k exactly one time.

So τ can be divided into two paths: a path from v_i to v_k which use only nodes $v_1, \ldots, v_{k-1}$ (the minimum length of such a path is equal to $d_{ik}^{(k-1)}$) and a path from v_k to v_j which use only nodes $v_1, \ldots, v_{k-1}$ (the minimum length of such a path is equal to $d_{kj}^{(k-1)}$).

Therefore, the length of τ is equal to $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$, and the considered equality holds.

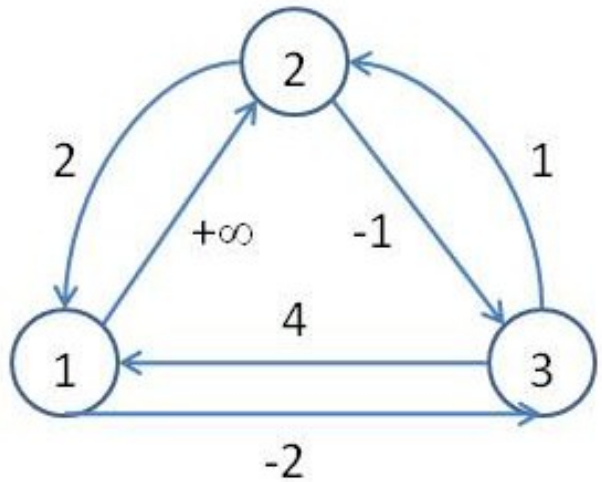
Shortest Paths

If we know $D^{(k-1)}$ then to compute $D^{(k)}$ it is enough to make n^2 operations of addition and n^2 operations of comparison of numbers.

Therefore, to compute $D^{(n)} = D^*$ it is enough to make $O(n^3)$ operations of addition and comparison.

Shortest Paths

Example 4.2. Shortest paths in complete directed graph



For every i, j and $k > 0$, $d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}$.
If $i = k$ or $j = k$ then $d_{ij}^{(k)} = d_{ij}^{(k-1)}$.

$$D^{(0)} = \begin{array}{c|ccc} & 1 & 2 & 3 \\ 1 & 0 & +\infty & -2 \\ 2 & 2 & 0 & -1 \\ 3 & 4 & 1 & 0 \end{array} \qquad D^{(1)} = \begin{array}{c|ccc} & 1 & 2 & 3 \\ 1 & 0 & +\infty & -2 \\ 2 & 2 & 0 & -1 \\ 3 & 4 & 1 & 0 \end{array}$$

$$D^{(2)} = \begin{array}{c|ccc} & 1 & 2 & 3 \\ 1 & 0 & +\infty & -2 \\ 2 & 2 & 0 & -1 \\ 3 & 3 & 1 & 0 \end{array} \qquad D^{(3)} = \begin{array}{c|ccc} & 1 & 2 & 3 \\ 1 & 0 & -1 & -2 \\ 2 & 2 & 0 & -1 \\ 3 & 3 & 1 & 0 \end{array} = D^*$$

Edit Distance

Let we have two strings X and Y in some alphabet. A natural measure of a distance between these strings is the degree to which they can be aligned, or matched up. An alignment is a way of writing the strings one above the other.

Let us consider two possible alignments of SNOWY and SUNNY

S	_	N	O	W	Y		_	S	N	O	W	_	Y
S	U	N	N	_	Y		S	U	N	_	_	N	Y

Edit Distance

The symbol "_" indicates a "gap". Any number of gaps can be added to each string. The cost of an alignment is the number of columns in which the letters differ. The *edit distance* between two strings is the cost of the best alignment.

In the first alignment the cost is equal to 3, and in the second alignment the cost is equal to 5.

In other words, the edit distance is the minimum number of insertions, deletions and substitutions of characters (letters) needed to transform the first string into the second one. In the first example we insert U, substitute O $\rightarrow$ N and delete W.

Edit Distance

Let $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$. For each $i \in \{0, 1, \ldots, m\}$ and $j \in \{0, 1, \ldots, n\}$, we consider the problem of finding of the edit distance between $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$, and denote by $E(i, j)$ the considered distance.

If $i = 0$, the word $x_1, \ldots, x_i$ is the empty word λ. The same situation is for the case $j = 0$. It is clear that $E(i, 0) = i$ and $E(0, j) = j$ since the edit distance between the empty word λ and a nonempty word α of the length t is equal to t.

Edit Distance

Let us consider the best alignment for $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$ where $i > 0$ and $j > 0$. It is clear that in the rightmost column we can have one of the following three things:

$$\begin{array}{ccc} x_i & - & x_i \\ - & y_j & y_j \end{array}$$

Edit Distance

In the first case, $E(i,j) = 1 + E(i-1,j)$.

In the second case, $E(i,j) = 1 + E(i,j-1)$.

In the third case, $E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1)$, where $\mathrm{diff}(i,j) = 0$ if $x_i = y_j$ and $\mathrm{diff}(i,j) = 1$ if $x_i \neq y_j$.

Therefore

$$E(i,j) = \min\{1+E(i-1,j), 1+E(i,j-1), \mathrm{diff}(i,j)+E(i-1,j-1)\}.$$

Edit Distance

We have $(m+1)(n+1)$ subproblems. If we know $E(i-1,j)$, $E(i,j-1)$, and $E(i-1,j-1)$ then to compute the value $E(i,j)$ it is necessary to make 3 operations of comparisons of numbers (1 to find the value $\mathrm{diff}(i,j)$, and 2 to find min) and 3 operations of addition.

So, the considered algorithm makes $O(mn)$ operations of addition and comparison of numbers.

Edit Distance

Really, to find the value $E(m, n)$ we should fill the table with $m + 1$ rows labeled with numbers $0, 1, \ldots, m$, and $n + 1$ columns labeled with numbers $0, 1, \ldots, n$. At the intersection of i-th row and j-th column we should have the number $E(i, j)$.

At the beginning, we can fill values $E(i, 0) = i$ and $E(0, j) = j$, and after that row by row, from the left to the right we can fill out the table.

Note that it is not necessary to have the whole table in the memory: to fill the i-th row, $i > 0$, it is enough to know values in the row $i - 1$.

Edit Distance

Now we can find the optimal alignment if the considered table is filled.

If $E(m, n) = 1 + E(m-1, n)$, then in the optimal alignment the last column is $\begin{matrix} x_m \\ - \end{matrix}$.

If $E(m, n) = 1 + E(m, n-1)$, then in the optimal alignment the last column is $\begin{matrix} - \\ y_n \end{matrix}$.

If $E(m, n) = \mathrm{diff}(m, n) + E(m-1, n-1)$, then in the optimal alignment the last column is $\begin{matrix} x_m \\ y_n \end{matrix}$.

Edit Distance

To find the next column we should consider:

In the first case – the subproblem $E(m-1, n)$.

In the second case – the subproblem $E(m, n-1)$.

In the third case – the subproblem $E(m-1, n-1)$, etc.

Edit Distance

Example 4.3. Edit distance between SUNNY and SNOWY

		S	U	N	N	Y
	0	1	2	3	4	5
S	1	**0**	**1**	2	3	4
N	2	1	**1**	**1**	2	3
O	3	2	2	**2**	**2**	3
W	4	3	3	3	**3**	3
Y	5	4	4	4	4	**3**

x_i y_j		x_i -
	i -1,j - 1	$i-1,j$
- y_j	i,j - 1	i,j

S	N	O	W	Y	
S	U	N	N	Y	
S	-	N	O	W	Y
S	U	N	N	-	Y
S	-	N	O	W	Y
S	U	N	-	N	Y

$$E(i,j) = \min\{1+E(i-1,j), 1+E(i,j-1), \mathrm{diff}(i,j)+E(i-1,j-1)\}.$$

Sequence Alignment

Let us consider a generalization of edit distance which is widely used in biology.

For the case of edit distance, the cost of matching a letter x_i or y_j and gap is equal to 1, and the cost of matching of x_i and y_j is equal to 0 if $x_i = y_j$ and 1 otherwise. The cost of alignment is the total cost of matchings.

Sequence Alignment

In general case, the cost of matching of x_i or y_j and gap is equal to $\delta > 0$. The cost of matching of x_i and y_j is equal to $\alpha(x_i, y_j) \geq 0$, and $\alpha(x_i, y_j) = 0$ if $x_i = y_j$. The cost of alignment is the total cost of matchings.

Sequence Alignment

We will use the notation $F(i,j)$ for the minimal cost of alignment for strings $x_1,\ldots,x_i$ and $y_i,\ldots,y_j$. It is clear that $F(i,0)=\delta i$ and $F(0,j)=\delta j$. One can show that for $i>0$ and $j>0$,

$$F(i,j)=\min\{\delta+F(i-1,j),\delta+F(i,j-1),\alpha(x_i,y_j)+F(i-1,j-1)\}.$$

Based on this formula, we can create an algorithm for computation of $F(m,n)$ which use $O(mn)$ operations of comparison and addition of numbers. This algorithm can be easily extended for construction of one of the best alignments.

Extensions of Dynamic Programming

Usual dynamic programming approach allows us to construct an optimal (relative to a cost function) object from a given set of objects.

We can extend this approach such that it will be possible to describe the whole set of optimal objects and even make sequential optimization relative to different cost functions.

We consider here an example connected with well known *maximum subarray problem*: for a given one-dimensional array of numbers, it is necessary to find a contiguous subarray which has the largest sum.

Optimization of Subarrays

Let $a_1, \ldots, a_n$ be an array (sequence) of integers. Contiguous subsequence of this sequence $a_i, \ldots, a_j$, $i \leq j$, is called a *subarray* of the considered array.

We study two parameters *sum* and *length* of subarrays defined in the following way: $sum(a_i, \ldots, a_j) = a_i + \ldots + a_j$ and $length(a_i, \ldots, a_j) = j - i + 1$.

We will consider sequential optimization of subarrays relative to these parameters: we need to describe all subarrays with maximum sum and among these subarrays to find ones with maximum length.

Graph G

First, we construct a graph G which describes the set S of all nonempty subarrays of the array $a_1, \ldots, a_n$.

To this end we will use two operations on sets A and B of subarrays: *concatenation* $A \times B = \{\alpha, \beta : \alpha \in A, \beta \in B\}$ and *union* $A \cup B$.

Graph G

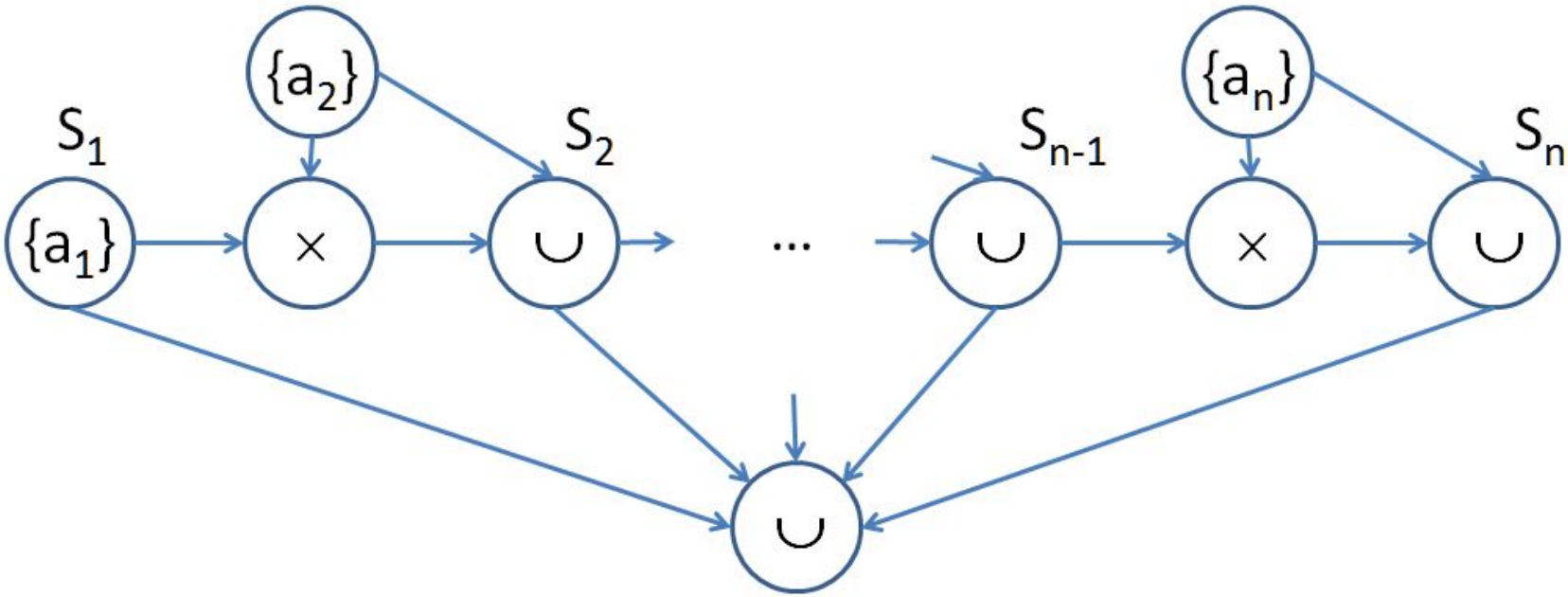
For $j = 1, \ldots, n$, we denote by S_j the set of all subarrays of the kind $a_i, \ldots, a_j$ where $i \leq j$. It is clear that $S_1 = \{a_1\}$ and

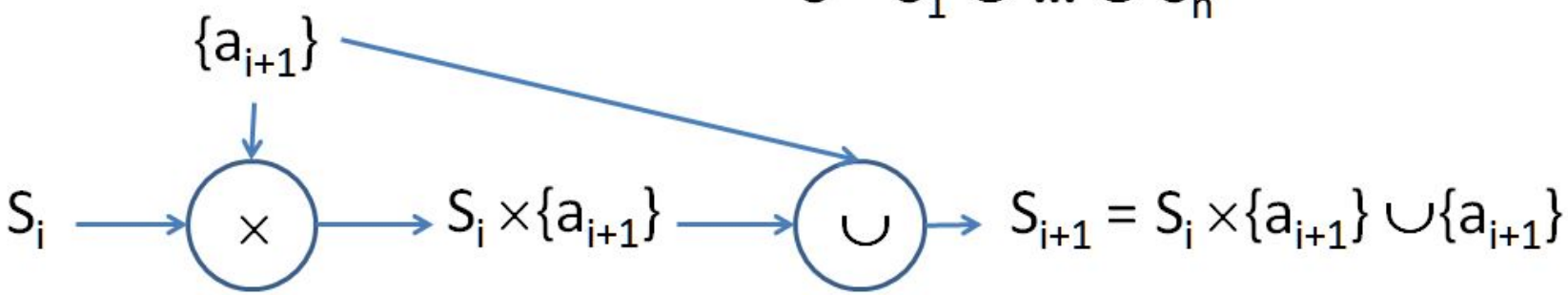
$$S_{j+1} = S_j \times \{a_{j+1}\} \cup \{a_{j+1}\}$$

for $j = 1, \ldots, n-1$.

It is clear also that $S = S_1 \cup \ldots \cup S_n$.

Graph G


$\{a_2\}$
$\{a_n\}$
S_1
S_2
S_{n-1}
S_n
$\{a_1\}$
×
∪
...
∪
×
∪
∪
$S = S_1 \cup \ldots \cup S_n$

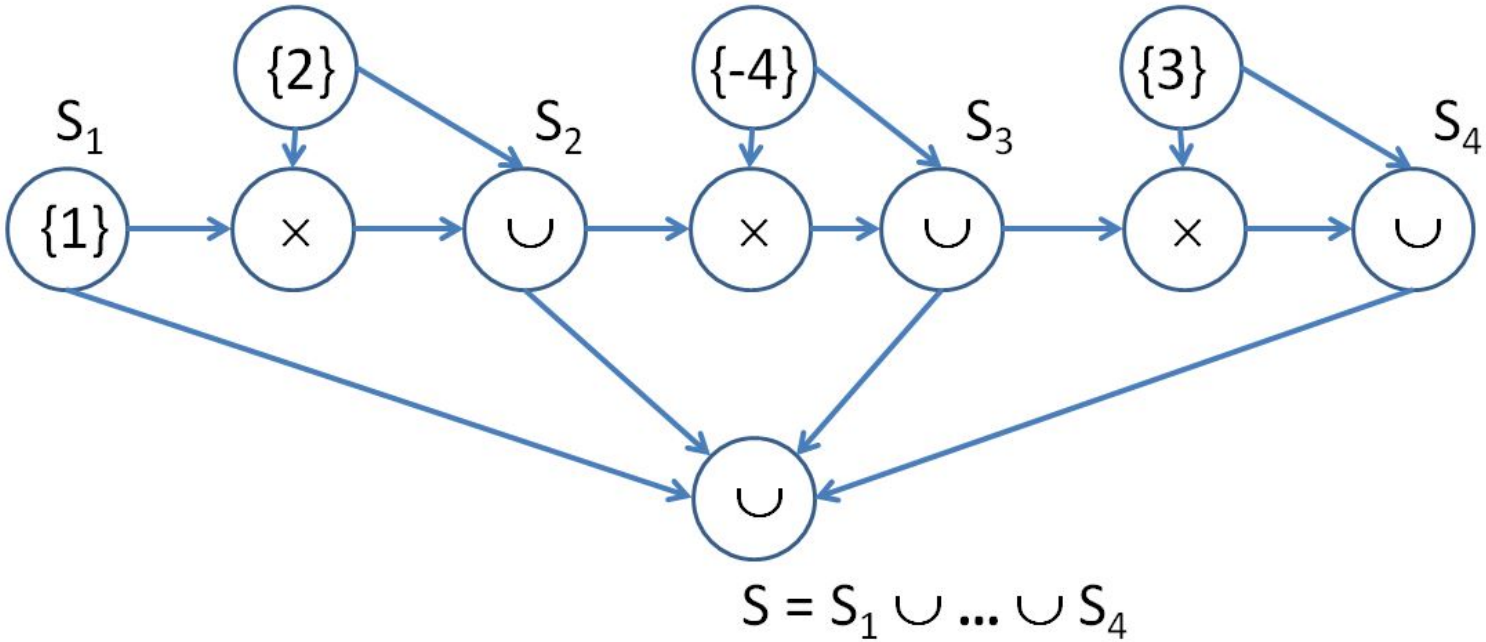

$\{a_{i+1}\}$
S_i
×
$S_i \times \{a_{i+1}\}$
∪
$S_{i+1} = S_i \times \{a_{i+1}\} \cup \{a_{i+1}\}$

Graph G

The graph G is a kind of circuit over the basis $\{\times, \cup\}$ which has one-element sets $\{a_1\}, \ldots, \{a_n\}$ at the inputs (all nodes on the first level of G and the first node one the second level) and the set S at the output (the only node on the third level of G).

We (in a natural way) can correspond to each node v of the graph G a subset $S_G(v)$ of the set S which is constructed in this node.

Example



Graph G for the array $1, 2, -4, 3$. For this array,
$S_1 = \{1\}$,
$S_2 = \{1, 2; 2\}$,
$S_3 = \{1, 2, -4; 2, -4; -4\}$,
$S_4 = \{1, 2, -4, 3; 2, -4, 3; -4, 3; 3\}$.

Optimization Relative to *sum*

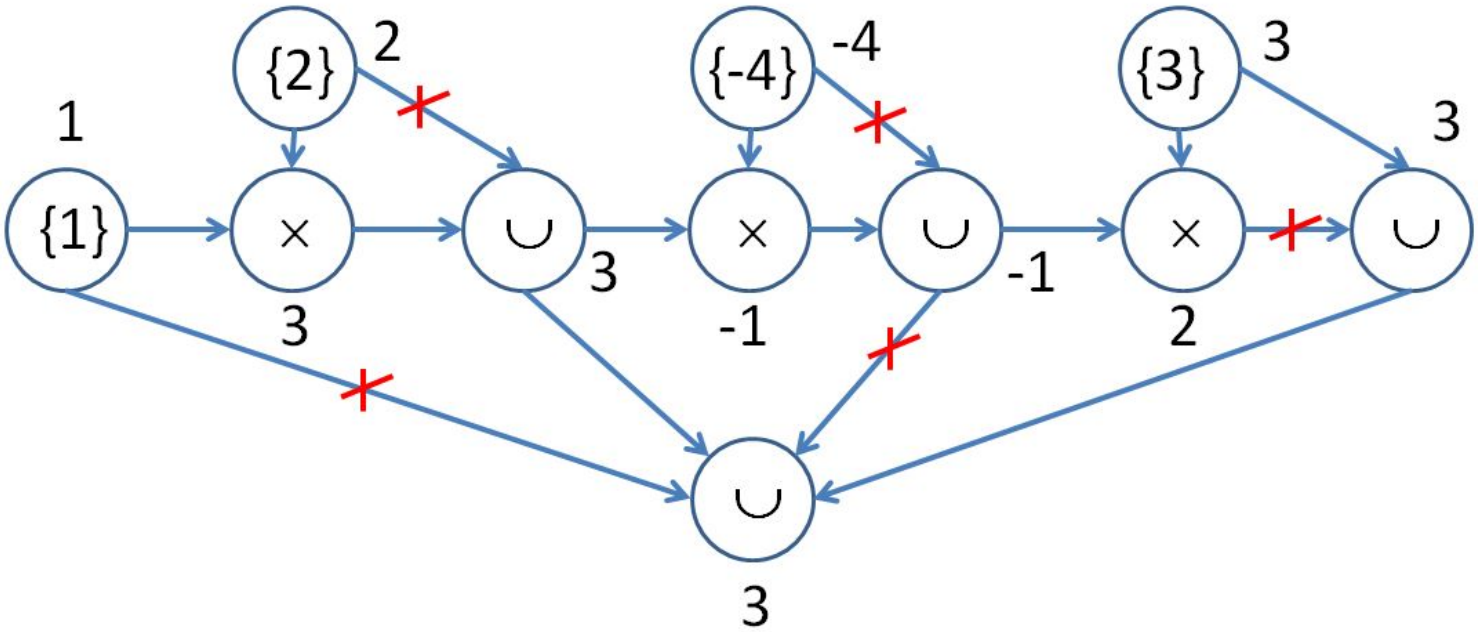
We consider the procedure of optimization of G relative to the cost function *sum*. We attach to each node v of G the number $sum(G, v) = \max\{sum(\alpha) : \alpha \in S_G(v)\}$ and, probably, remove some edges that enter v.

If v is an input node $\{a_i\}$ then $sum(G, v) = a_i$.

If v is a $\times$-node with two parents v_1 and v_2 then $sum(G, v) = sum(G, v_1) + sum(G, v_2)$.

If v is a $\cup$-node with k parents $v_1, \ldots, v_k$ then $sum(G, v) = \max\{sum(G, v_1), \ldots, sum(G, v_k)\}$. For $i = 1, \ldots, k$, if $sum(G, v) > sum(G, v_i)$, then we remove the edge from v_i to v.

Example (continuation)



Optimization relative to *sum* for the array $1, 2, -4, 3$.

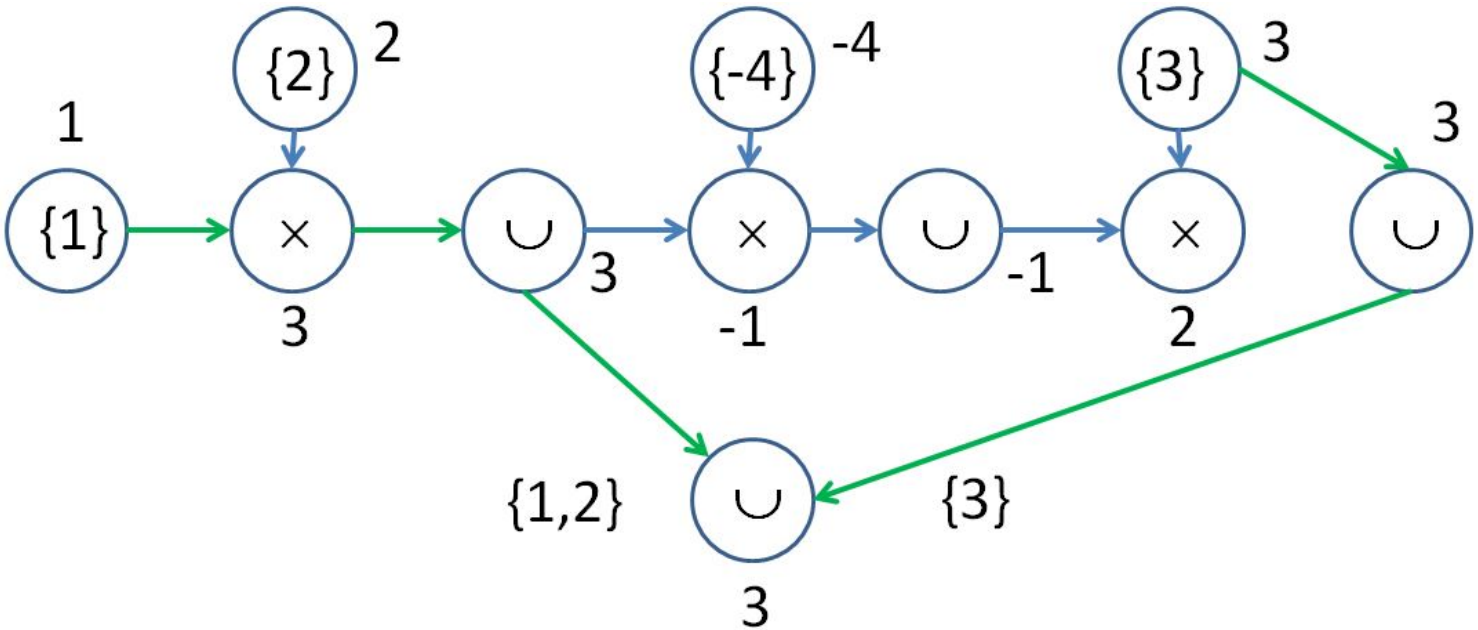
Optimization Relative to *sum*

As a result, we have a new graph G_1 (circuit) obtained from G by removal some edges.

We (in a natural way) can correspond to each node v of the graph G_1 a subset $S_{G_1}(v)$ of the set S which is constructed in this node.

One can show that, for each node v of G, the set $S_{G_1}(v)$ coincides with the set of all subarrays from $S_G(v)$ which have maximum sum among all subarrays from $S_G(v)$.

Example (continuation)



Graph G_1 for the array $1, 2, -4, 3$. There are two subarrays with the maximum sum: $1, 2$ and 3.

Optimization Relative to *length*

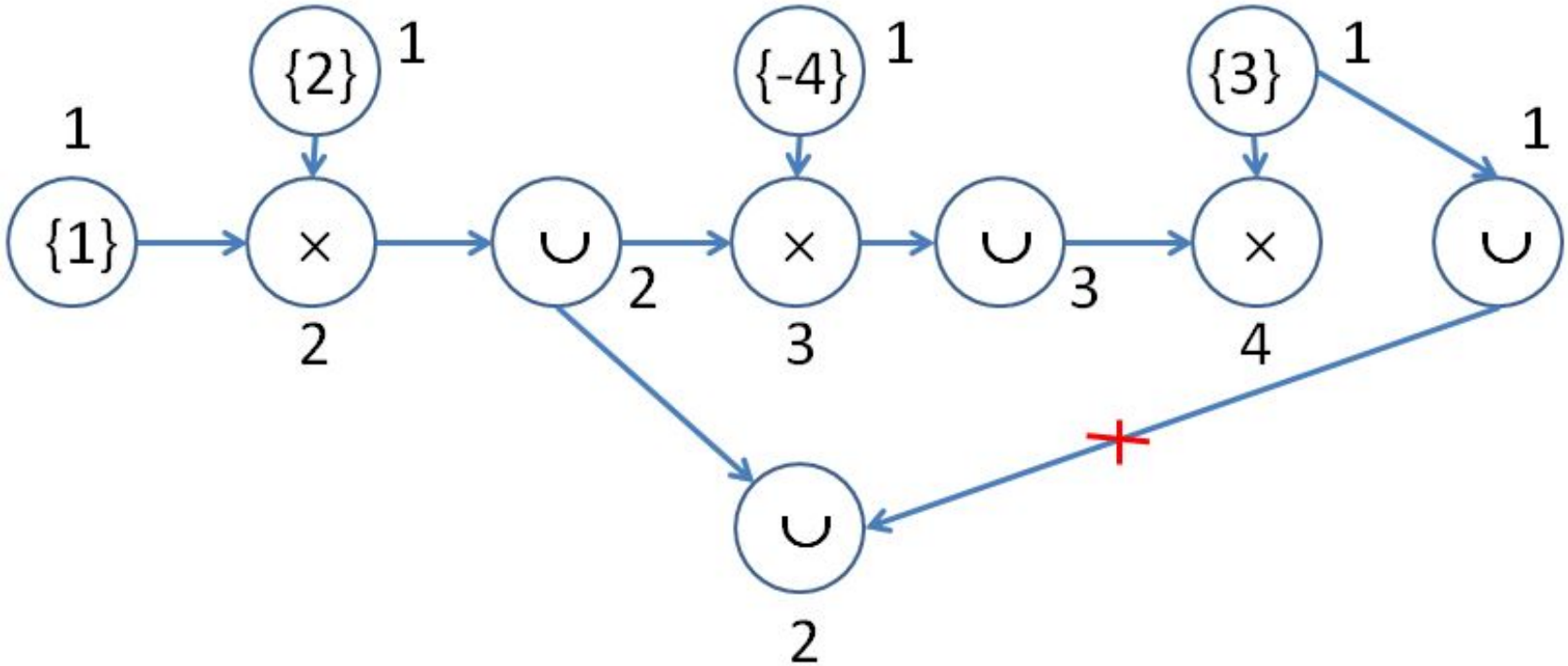
We now consider the procedure of optimization of G_1 relative to the cost function *length*. We attach to each node v of G_1 the number $length(G_1, v) = \max\{length(\alpha) : \alpha \in S_{G_1}(v)\}$ and, probably, remove some edges that enter v.

If v is an input node $\{a_i\}$ then $length(G_1, v) = 1$.

If v is a $\times$-node with two parents v_1 and v_2 then $length(G_1, v) = length(G_1, v_1) + length(G_1, v_2)$.

If v is a $\cup$-node with k parents $v_1, \ldots, v_k$ then $length(G_1, v) = \max\{length(G_1, v_1), \ldots, length(G_1, v_k)\}$. For $i = 1, \ldots, k$, if $length(G_1, v) > length(G_1, v_i)$, then we remove the edge from v_i to v.

Example (continuation)



Optimization relative to *length* for the array $1, 2, -4, 3$ and graph G_1.

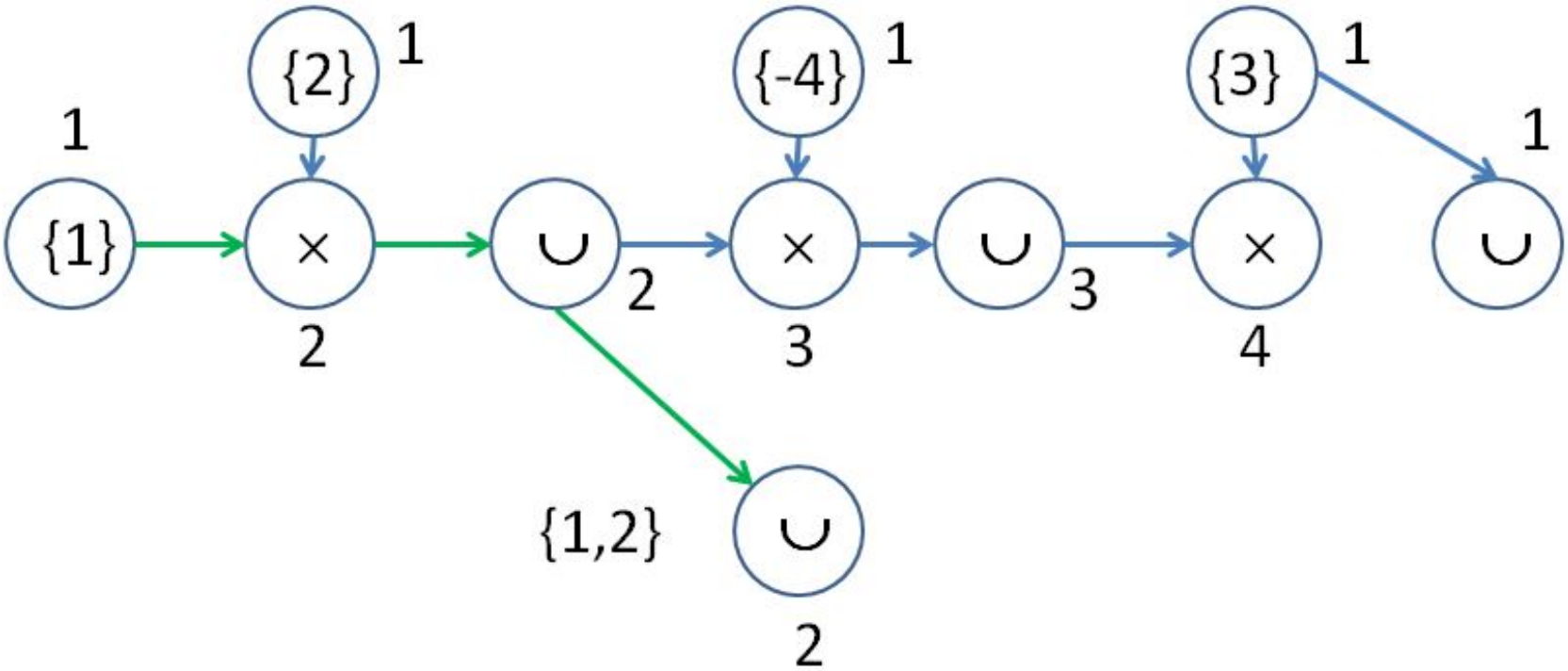
Optimization Relative to *length*

As a result, we have a new graph G_2 (circuit) obtained from G_1 (and from G) by removal some edges.

We (in a natural way) can correspond to each node v of the graph G_2 a subset $S_{G_2}(v)$ of the set S which is constructed in this node.

One can show that, for each node v of G, the set $S_{G_2}(v)$ coincides with the set of all subarrays from $S_{G_1}(v)$ which have maximum length among all subarrays from $S_{G_1}(v)$.

Example (continuation)



Graph G_2 for the array $1, 2, -4, 3$. There is only one subarray with the maximum length among all subarrays with the maximum sum: $1, 2$.

Complexity of Sequential Optimization

The graph G contains $3n - 1$ nodes.

One can show that the time complexity of construction of the graphs G, G_1, and G_2 is $O(n)$.