ALG 10

Dynamic programming

abbreviation: DP

Sources, overviews, examples see

https://cw.fel.cvut.cz/wiki/courses/ae4b33alg/links

Dynamic programming

DP **is a general strategy applicable to many different optimisation problems in diverse fields of computer science.**
In this respect it is similar to Divide and conquer strategy.

Important properties

1. The desired optimal solution is composed of suitably chosen optimal solutions of the same problem with reduced data.

2. The recursive formulation of the solution depends on many identical and repeated subproblems.
DP avoids unnecessary repeated computations by the method of tabelation (memoization) of the results of the subproblems.

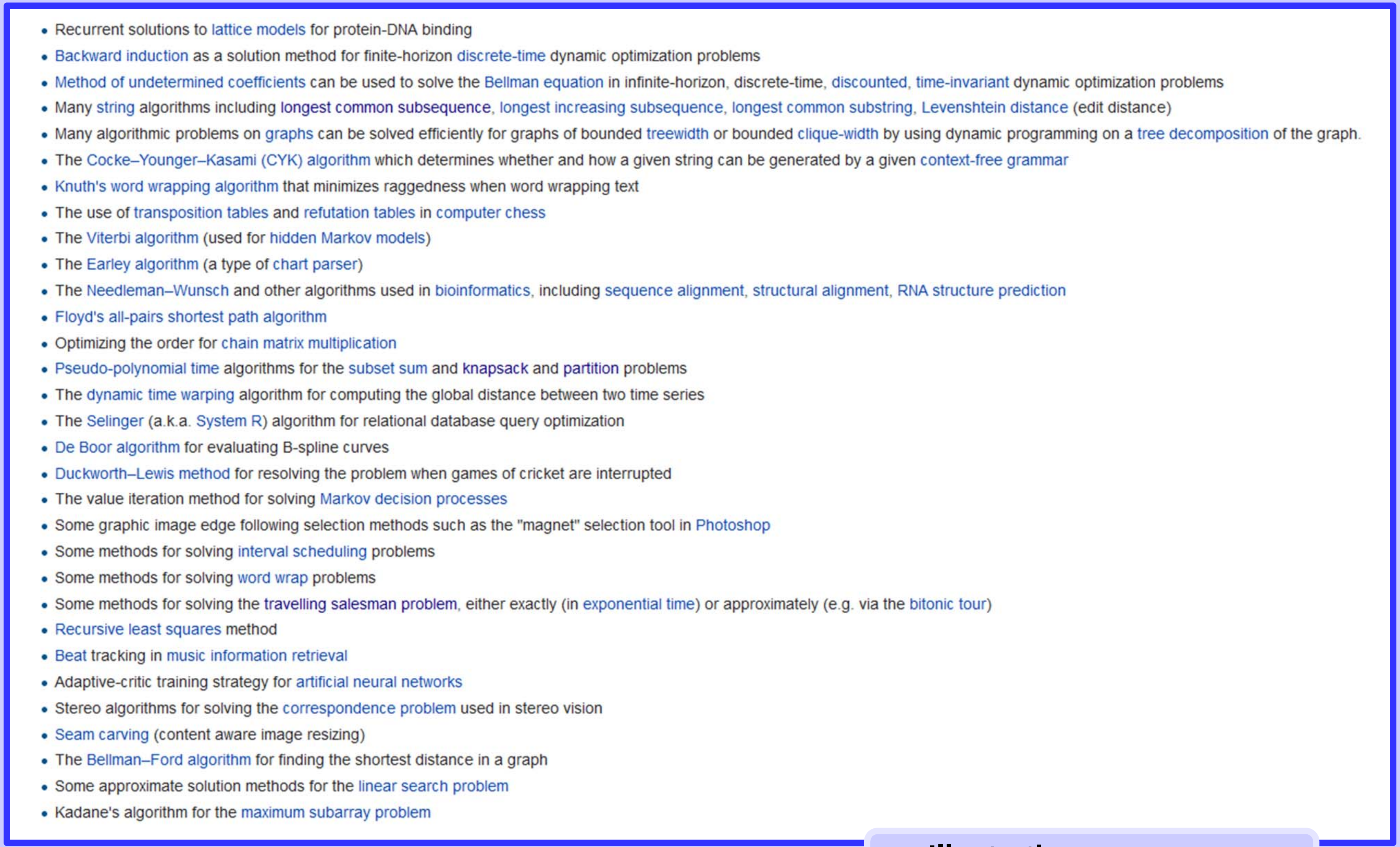
Dynamic programming

Application examples:

- **Optimal paths in graphs**
- **Longest subsequences with prescribed properties**
- **Knapsack problem**
- **Optimal scheduling of interdependent processes**
- **Approximate matching of patterns in text (bioinformatics)**
- **Longest common subsequence**
- **Optimal order of matrix multiplication**
- **Optimal binary search tree**
- **Optimal node/edge covering of a tree**
- **And many more....**

Dynamic programming

List of DP algorithms on en.wikipedia.org/wiki/Dynamic_programming



- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text
- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman–Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the subset sum and knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted
- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content aware image resizing)
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Some approximate solution methods for the linear search problem
- Kadane's algorithm for the maximum subarray problem

Illustrative screen copy

Tabelation in DP - example

Function definition

$$f(x,y) = \begin{cases} 1 & (x = 0) \ \ || \ \ (y = 0) \\ 2 \cdot f(x, y-1) + f(x-1,y) & (x > 0) \ \&\& \ (y > 0) \end{cases}$$

Problem

f(10,10) = ?

Program

```
def f(x, y):
    if (x == 0) or (y == 0): return 1
    return 2*f(x,y-1) + f(x-1,y)
```

```
print( f(10,10) );
```

Solution

f(10,10) = 127 574 017 ☺

Tabelation in DP - example

Simple analysis

```
count = 0

def f(x, y):
    global count
    count += 1
    if (x == 0) or (y == 0):
        return 1
    return 2*f(x,y-1) + f(x-1,y)
```

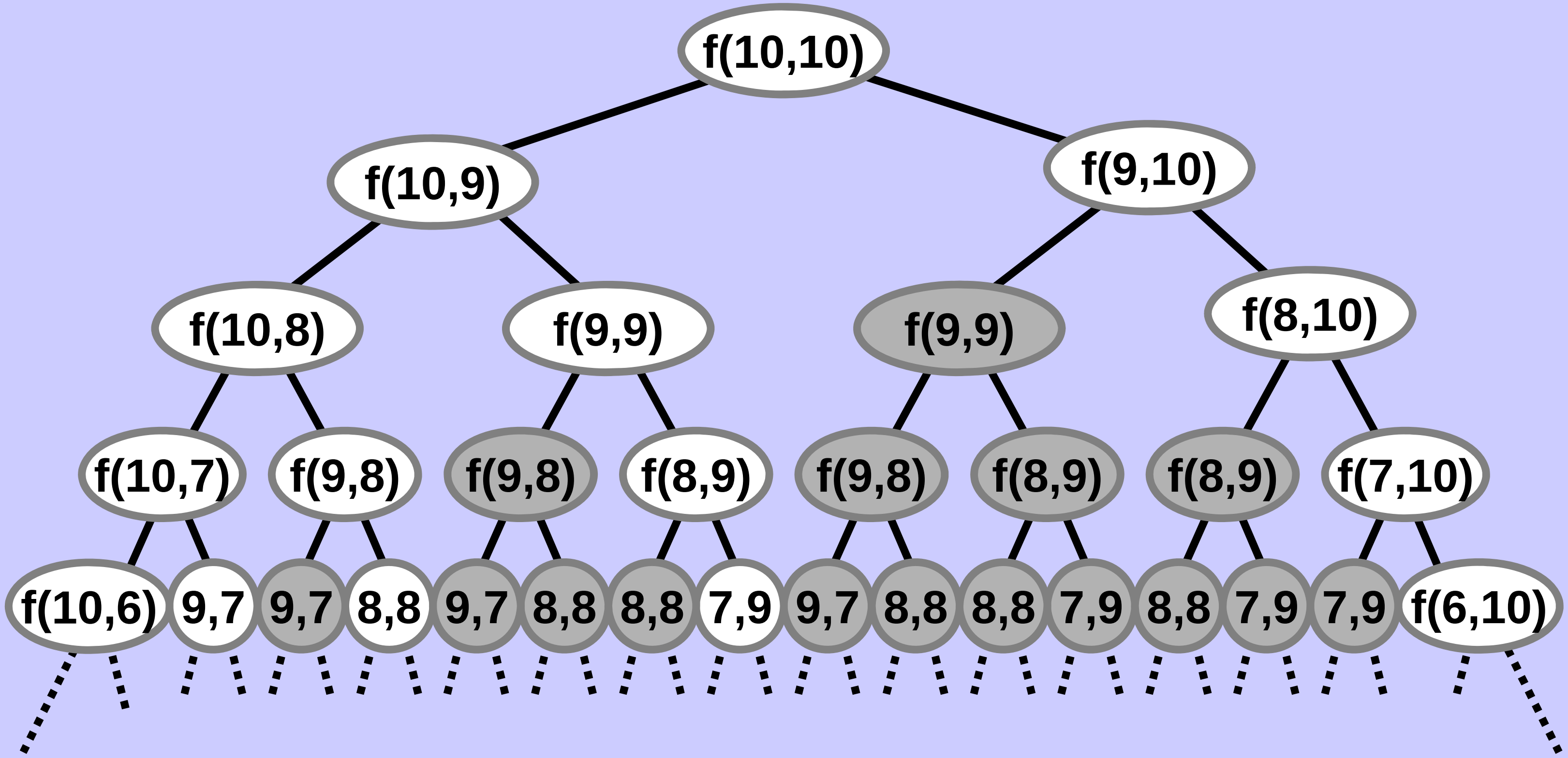
```
f(10,10)
print(count)
```

Analysis result

count = 369 511 ☹

Tabelation in DP - example

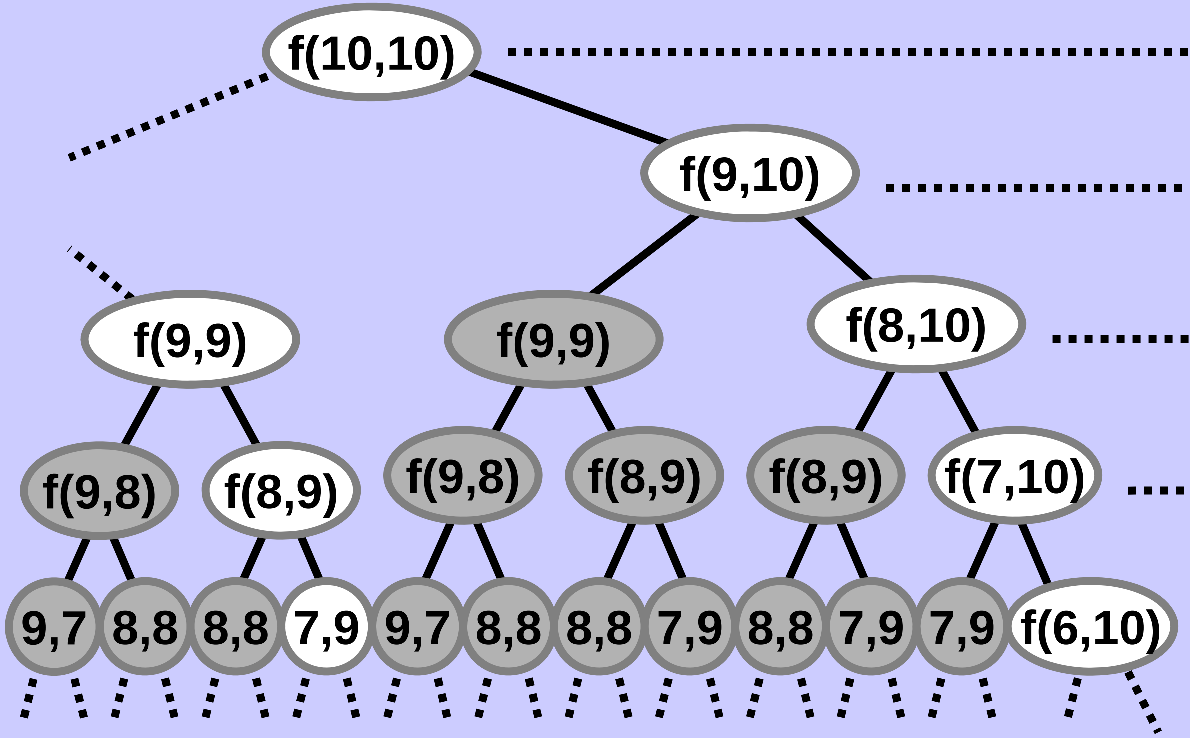
More detailed analysis – recursion tree



f(x,y) ... repetitive calculations, many of them!

Tabelation in DP - example

Detailed analysis continues – recursive calls effectivity

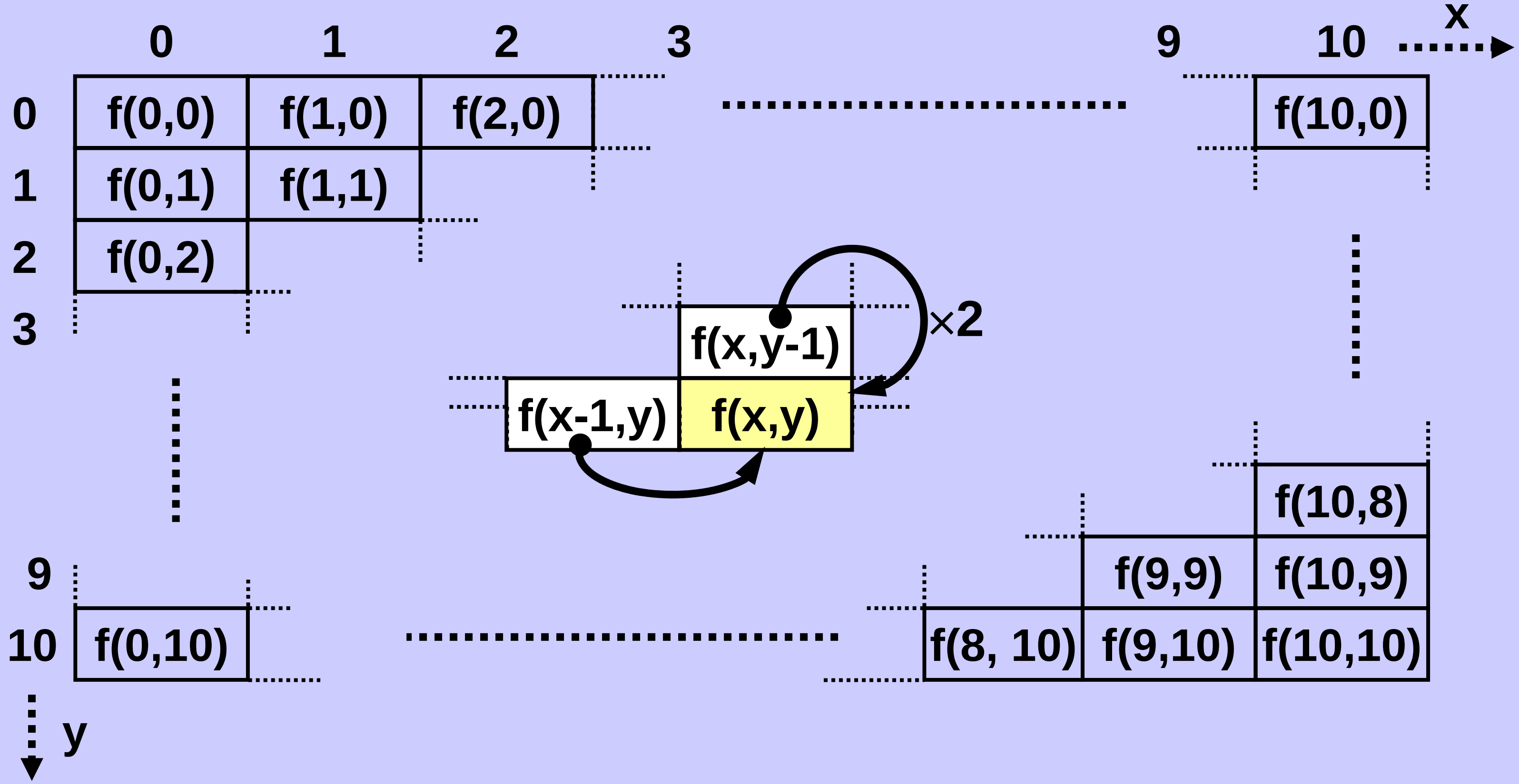


count:	calls		values
	1	....	1
	2	....	2
	4	....	3
	8	....	4
	16	....	5
	...		...
	512	☹	10
	...		...

Tabelation in DP - example

$$\boxed{f(x,y)} = \begin{cases} 1 & (x = 0) \;\; || \;\; (y = 0) \\ 2 \cdot \boxed{f(x, y-1)} + \boxed{f(x-1,y)} & (x > 0) \;\; \&\& \; (y > 0) \end{cases}$$

Table in general

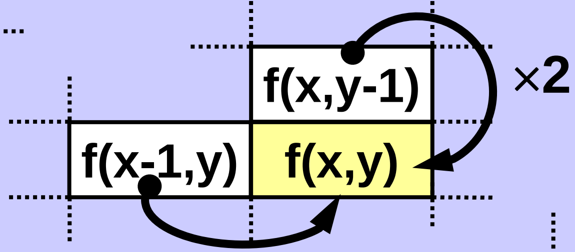


Tabelation in DP - example

$$\boxed{f(x,y)} = \begin{cases} 1 & (x = 0) \ \| \ (y = 0) \\ 2 \cdot \boxed{f(x, y-1)} + \boxed{f(x-1,y)} & (x > 0) \ \&\& \ (y > 0) \end{cases}$$

Table with numerical values

y \ x	0	1	2	3	4		9	10
0	1	1	1	1	1			1
1	1	3	5	7	9			
2	1	7	17	31				
3	1	15	49					
4	1	31						
								8085505
9							16807935	32978945
10	1					28000257	61616127	127574017



Tabelation in DP - example

All values are precomputed

```
def dynTable(M, N):
    # initialize by constant 1 for x = 0 or y = 0
    t = [[1]*(N+1) for i in range(M+1)]

    for y in range(1, M+1):
        for x in range(1, N+1):
            t[y][x] = 2*t[y-1][x] + t[y][x-1]

    return t[M][N]
```

Function call

```
def f(x, y):
  return dynTable(y, x)
```

Optimal paths in graphs

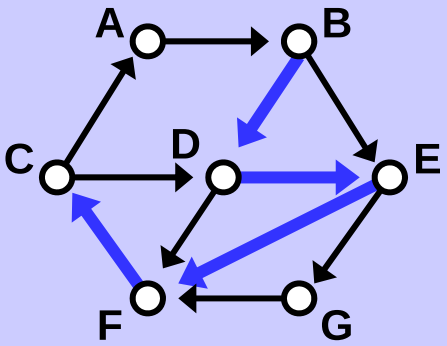
Notation:

Graph G = (V, E), set of nodes resp. edges: V(g) resp. E(G),
N = |V(G)|, M = |E(G)|, or n = |V|, m = |M| etc.

Path in a graph
= sequence of incident edges which contains each node
at most once.

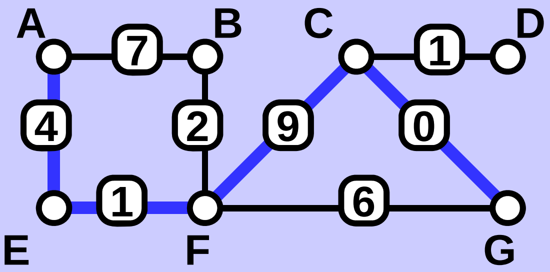
Path length in unweighted graph
= no. of edges in the path.

Ex. Length (B D E F C) = 4.



Path length in wighted graph
= sum of edge weights
in the path.

Ex. Length (A E F C G) = 14.



Optimal paths in graphs

Shortest paths

Problem of finding a shortest path between two given nodes or between more nodes or between all nodes in the graph. (E.g. Minimizing resources necessary to travel from x to y.)

Methods

The problem is solved for all practical cases of graphs.
We met earlier BFS which solves simple cases,
in more complex cases, particularly of weighted graphs,
specific algorithms are available -- Dijkstra, Floyd-Warshall,
Johnson, Bellman-Ford, etc.

Complexity

Asymptotic complexity is always polynomial in number of nodes and edges. Typically, the complexity of finding one path is at most $O(N^2)$, often less.

Optimal paths in graphs

Longest paths

Find a longest path between two given nodes or the longest path in the graph at whole.
(E.g. maximize the profit of temporary related processes.)

Exponential complexity

NP - hard problem

No systematic satisfactory solution has been found yet.

Possible strategies

1. Brute force -- exponential complexity, useless when N > cca 20.

2. Algorithms for approximate solutions with polynomial complexity
    -- either find optimum with limited probability < 1
    -- or can guarantee only a suboptimal solution
    -- are often non-trivial and requiring advanced impementation.

Optimal paths in graphs

Possible strategies

3. Some specific types of graph allow for application
of specific and effective algorithms (limited to that type of graph)

Most simple cases

3A.

Graph is a tree (both weighted or unweighted and both directed or undirected). Optimum path can be found in time $\Theta(N)$ by easy postorder traversal modification.

Opportunity for DP approach

3B.

Graph is directed and acyclic, may be weighted or unweighted.
Standard abbreviation: DAG (Directed Acyclic Graph)

Topological order of DAG nodes

Topological order od DAG is such ordering of its nodes
in which each edge points from the node with the lower order
to the node with the higher order.

Each DAG can be topologically ordered, usually in more ways.

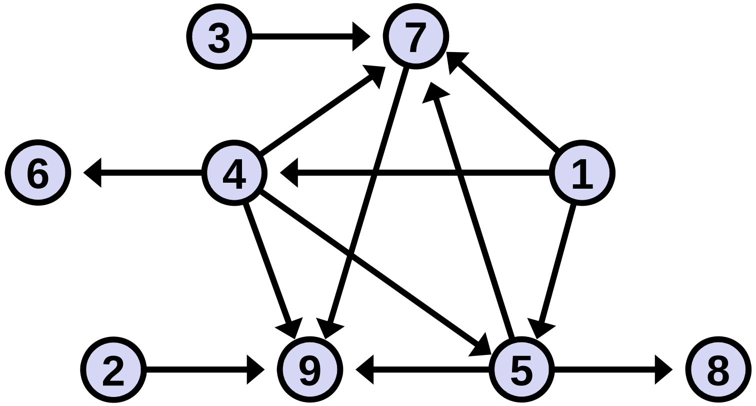
Any directed graph which contains at least one cycle
cannot be topologically ordered.

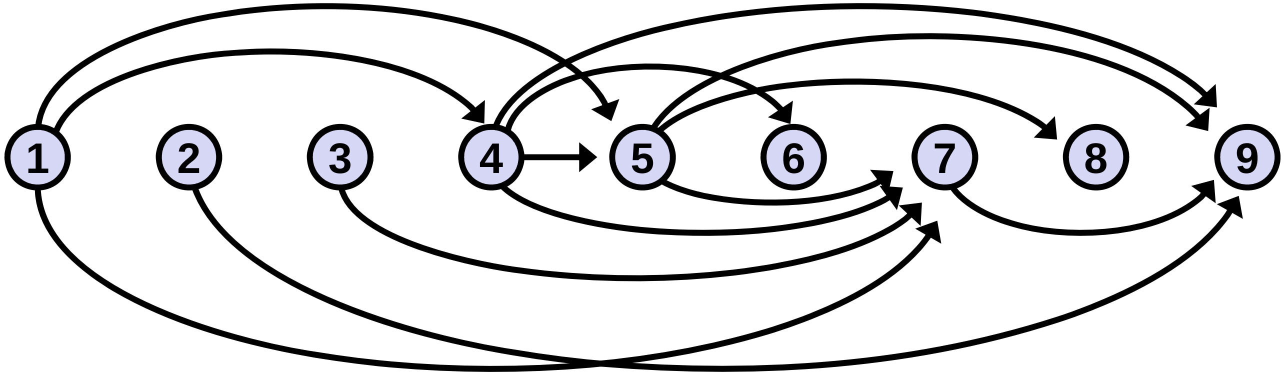
Graphs in some DP problems are implicitely or naturally
topologically ordered from the moment of problem posing.

Topological order of any DAG (at least one) can be found
in time Θ(M), i.e. in time proportional to the nuber of edges.

DAG and its topological order

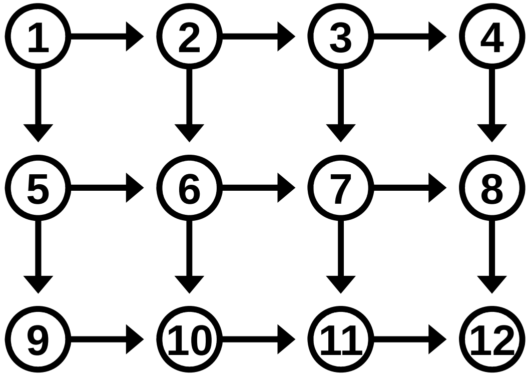
Example 1

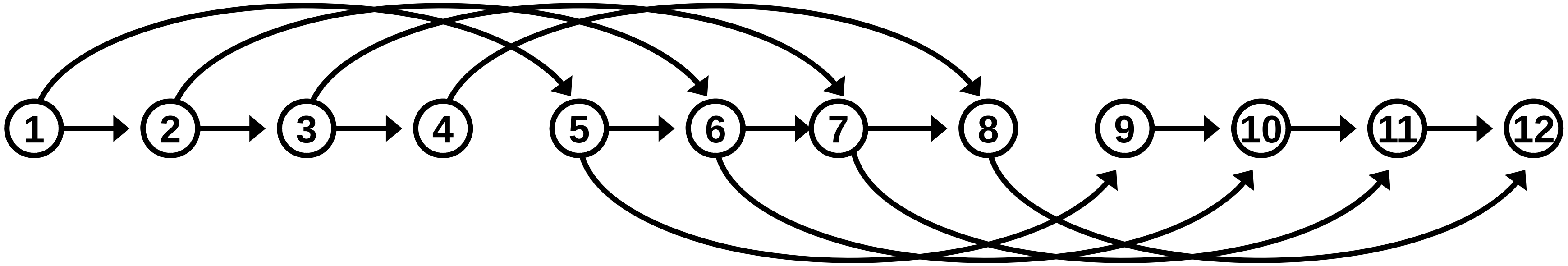




DAG and its various topological orders

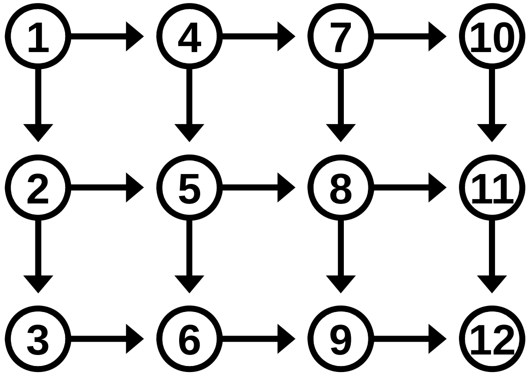
Example 2a

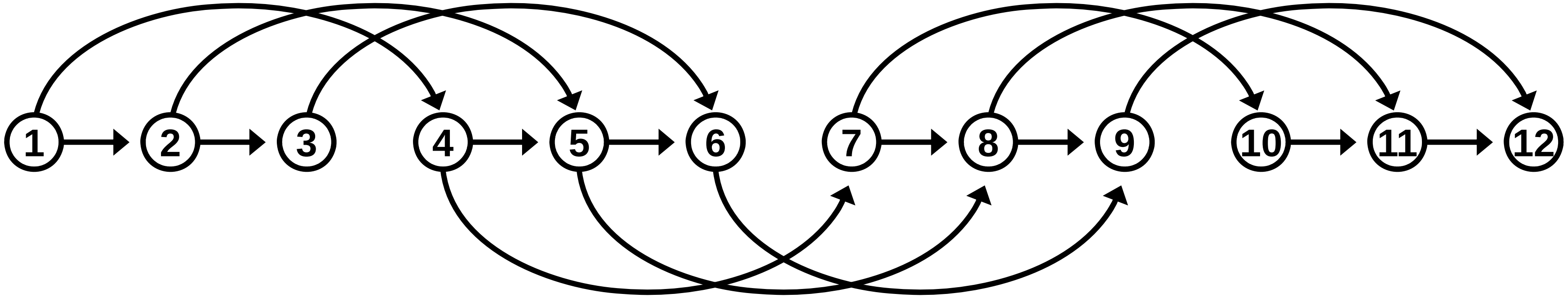




DAG and its various topological orders

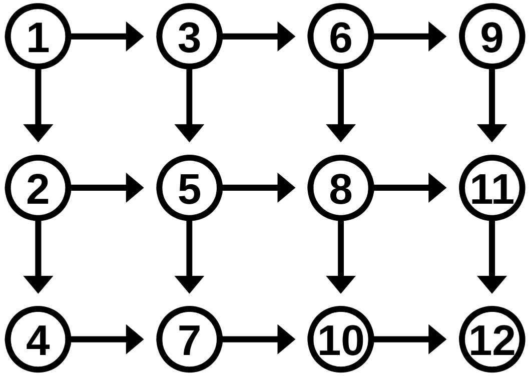
Example 2b

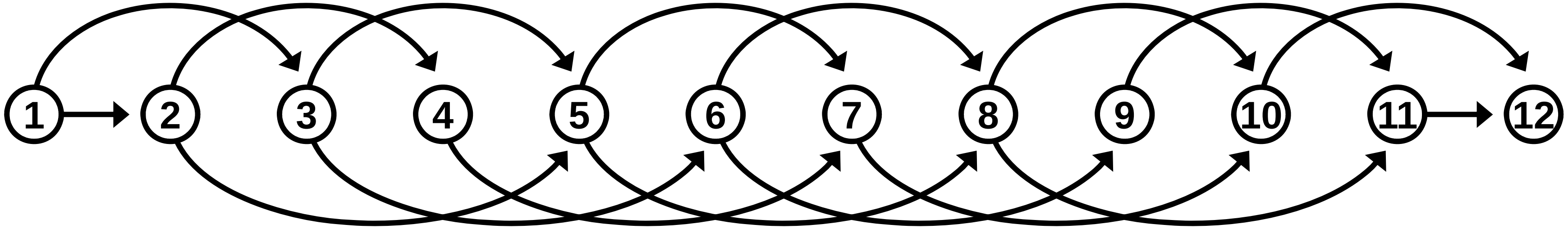




DAG and its various topological orders

Example 2c





Topological sort

Algorithm

```
0. new queue Q of Node

1. for each x in V(G):
       if x.indegree == 0:          # x is a root
          Q.insert(r)

2. while !Q.empty():
      v = Q.pop()
      for each edge (v, w) ∈ E(G):
          G.removeEdge((v, w))
          if w.indegree == 0:       # w is a root
               Q.insert(w)
```

Complexity

We suppose that operation
G.removeEdge((v, w))
has constant complexity[*)].

0. Complexity $O(N)$
1. Complexity $\Theta(N)$
2. Complexity $\Theta(M)$, each edge is visited exactly once and it is processed in constant time.

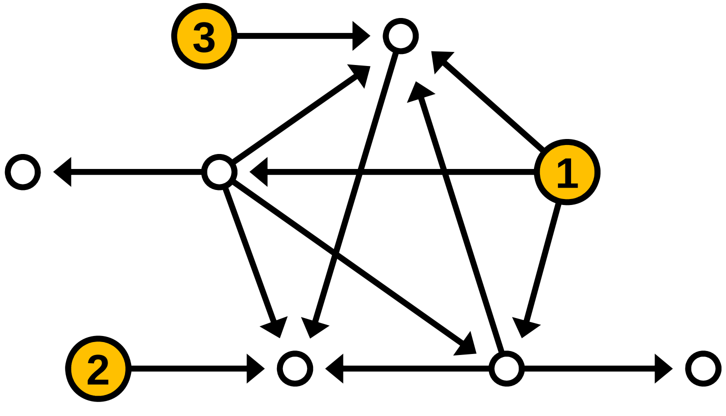
Complexity : $\Theta(N+M)$

Topological order

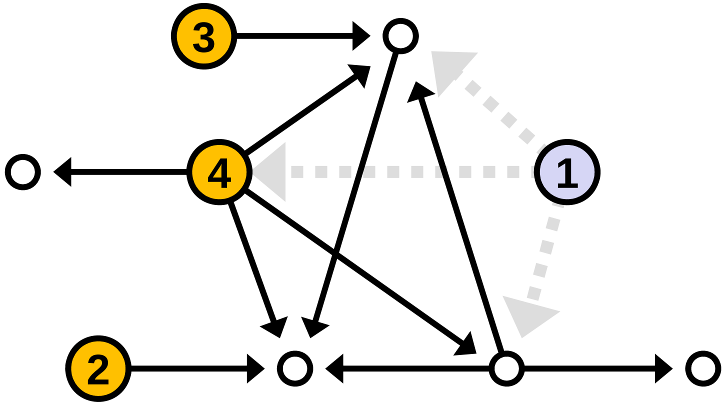
The order in which nodes are inserted into the queue is the topological order of DAG.

[*)]. Often it is enough to just *mark* the edge as deleted, without physically deleting it.

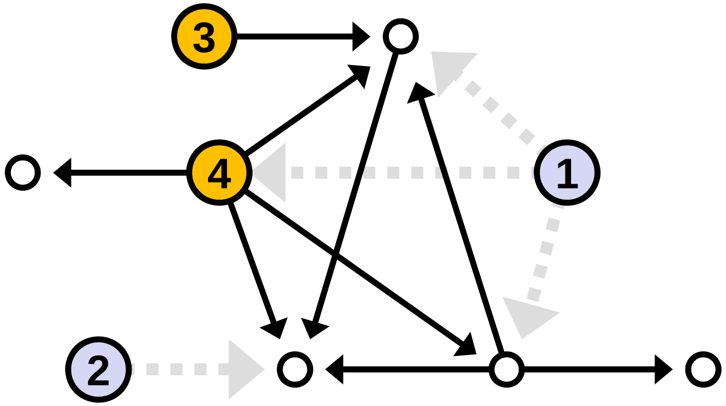
Topological sort -- example



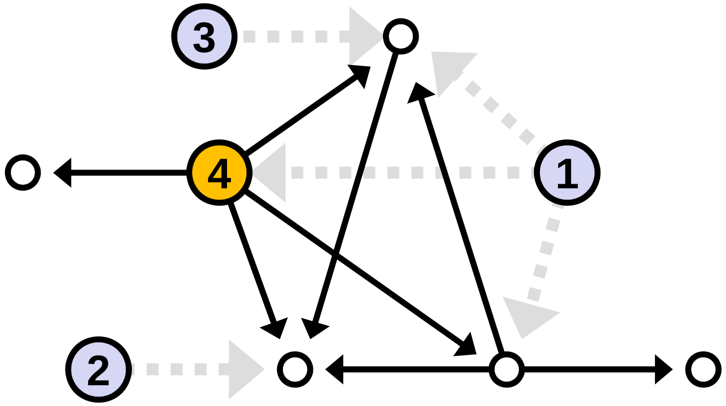
Queue: 1, 2, 3.



Queue: 2, 3, 4.

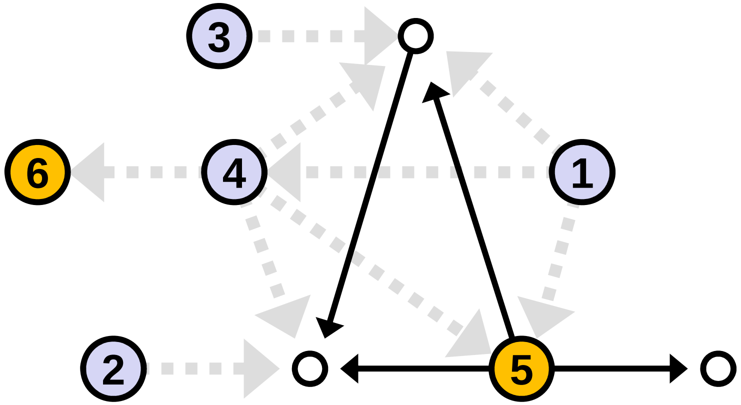


Queue: 3, 4.

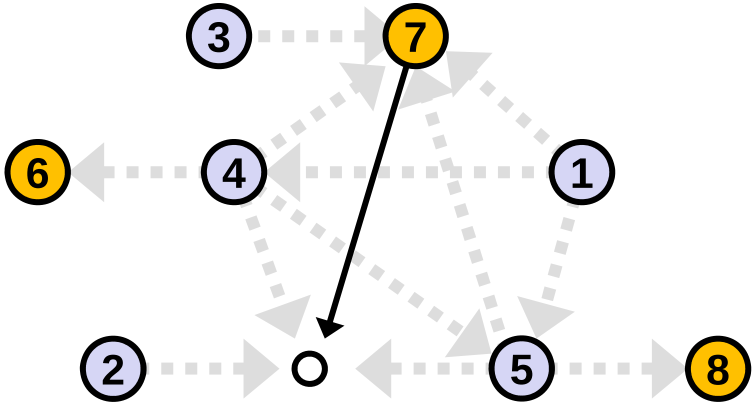


Queue: 4.

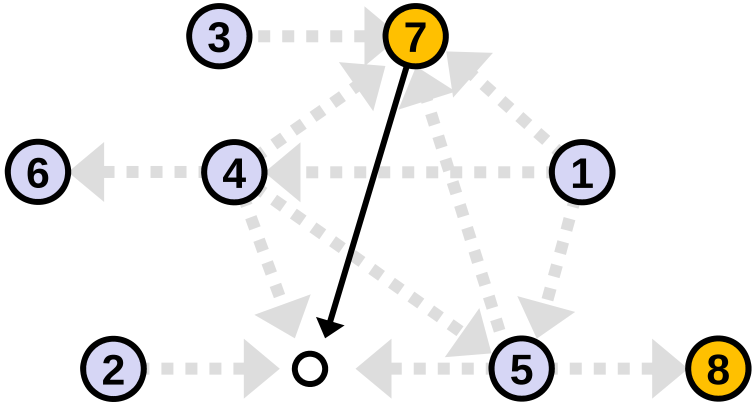
Topological sort -- example



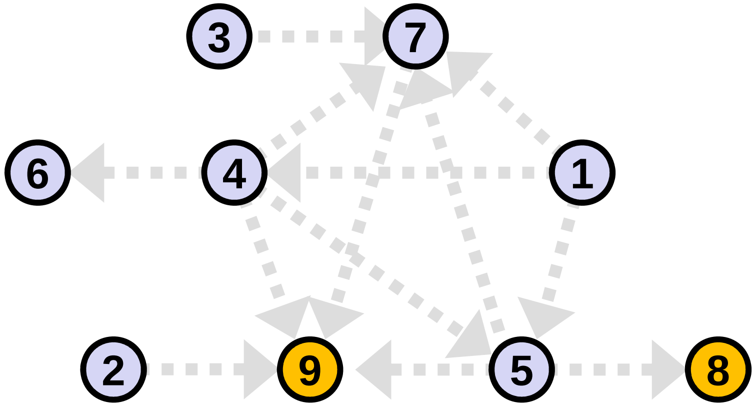
Queue: 5, 6.



Queue: 6, 7, 8.

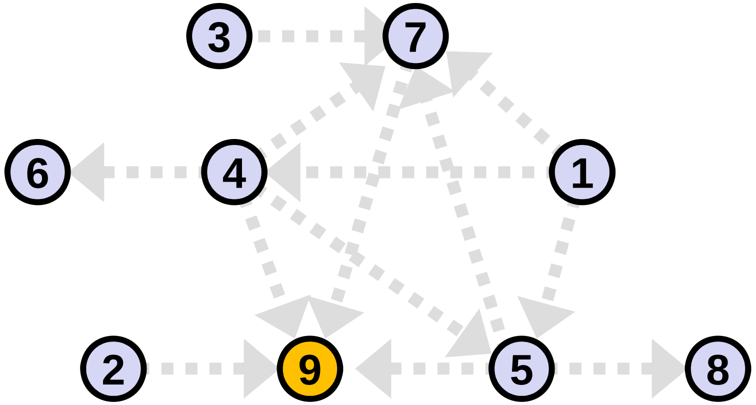


Queue: 7, 8.

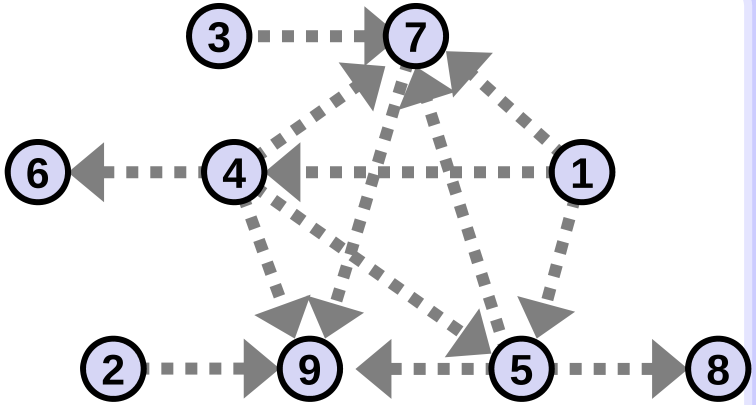


Queue: 8, 9.

Topological sort -- example

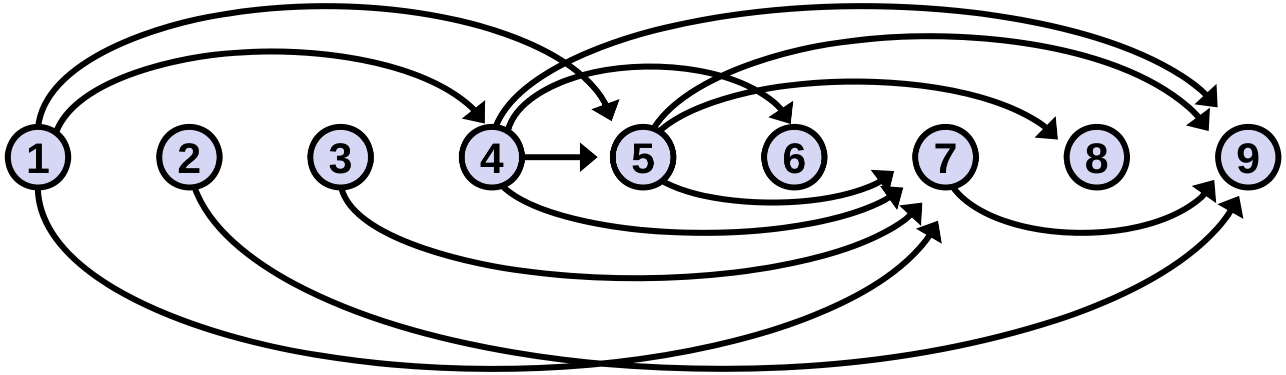


Queue: 9.



Queue: Empty.

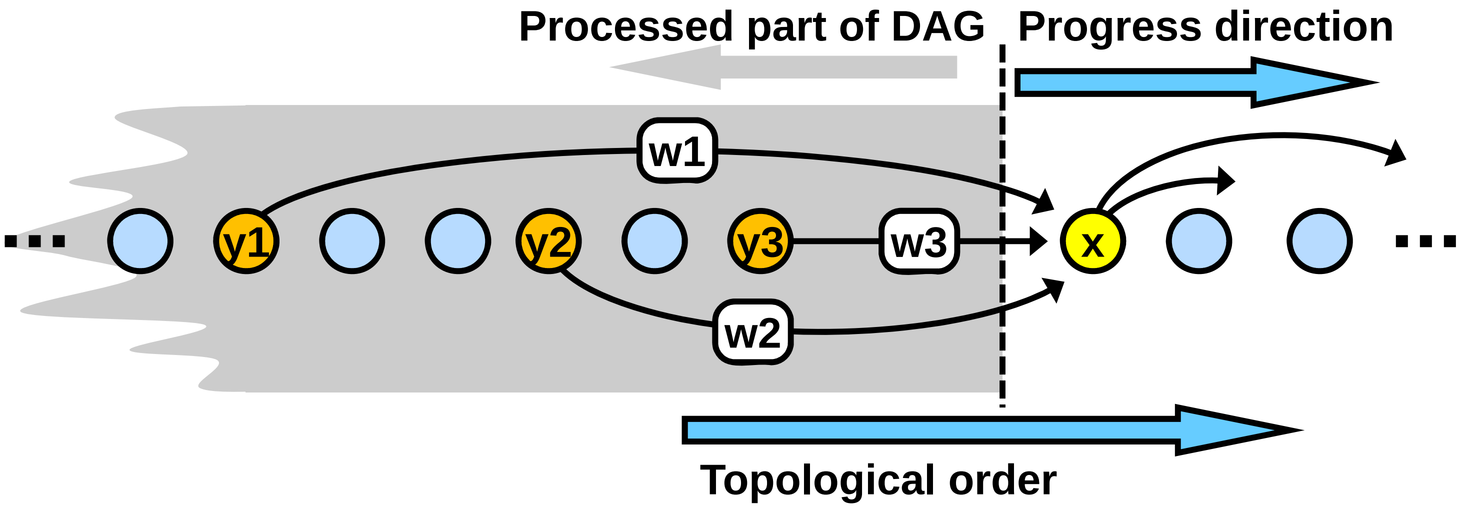
Topological order



Longest path in DAG

We process nodes of DAG in their topological order.
Denote by d[x] length of the path which ends in x and its length is maximal.

Charakteristic DP view "from the end to the beginning":
-- d[x] is set when values of d are known for all previous
(= already processed) nodes in the topological order.
-- d[x] is the maximum of values
{d[y1] + w1, d[y2] + w2, ..., d[yk] + wk},
where (y1, x), (y2, x) , ... are all edges ending in x
and w1, w2, ..., are their respective weights.

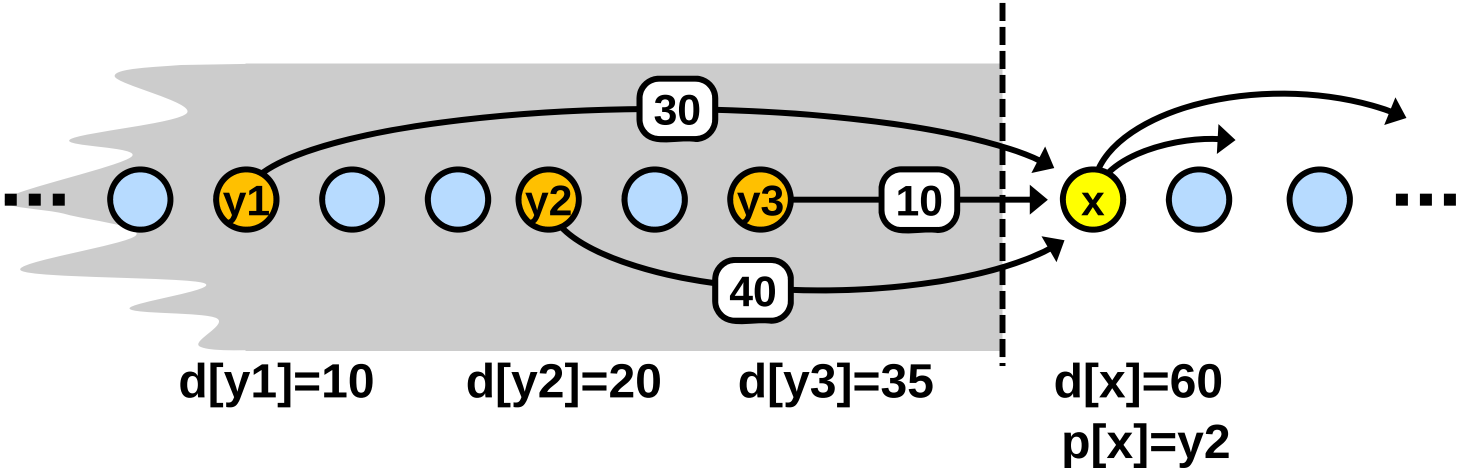


Longest path in DAG

-- d[x]is the maximum of values
{d[y1] + w1, d[y2] + w2, ..., d[yk] + wk},
where (y1, x), (y2, x) , ... are all edges ending in x
and w1, w2, ..., are their respective weights.

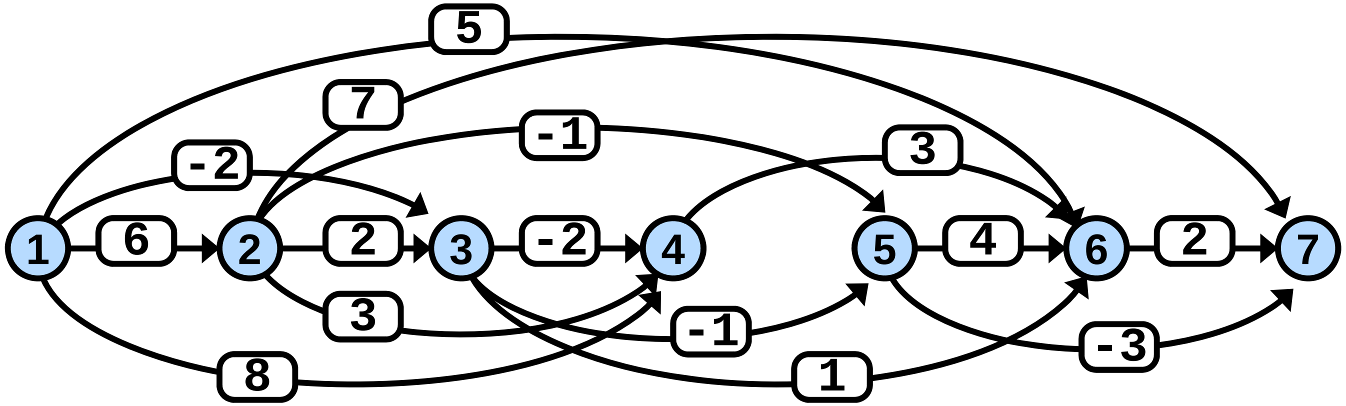
-- If all values {d[y1] + w1, d[y2] + w2, ..., d[yk] + wk}
are negative then none of them contributes to the longest path
and the value of d[x] is reset: d[x] = 0.

-- The node yj, for which the value d[yj] + wj is maximal and non-negative,
is set as a predecessor of x on the longest path.



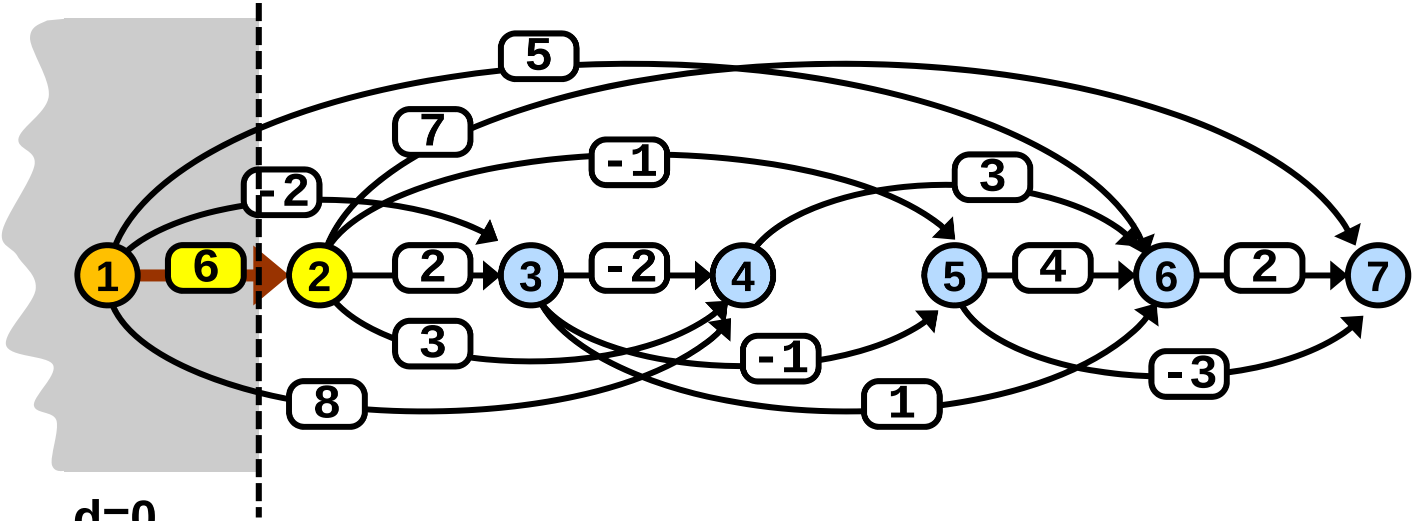
Longest path in DAG

Example



Find the longest path and its length.

Longest path in DAG



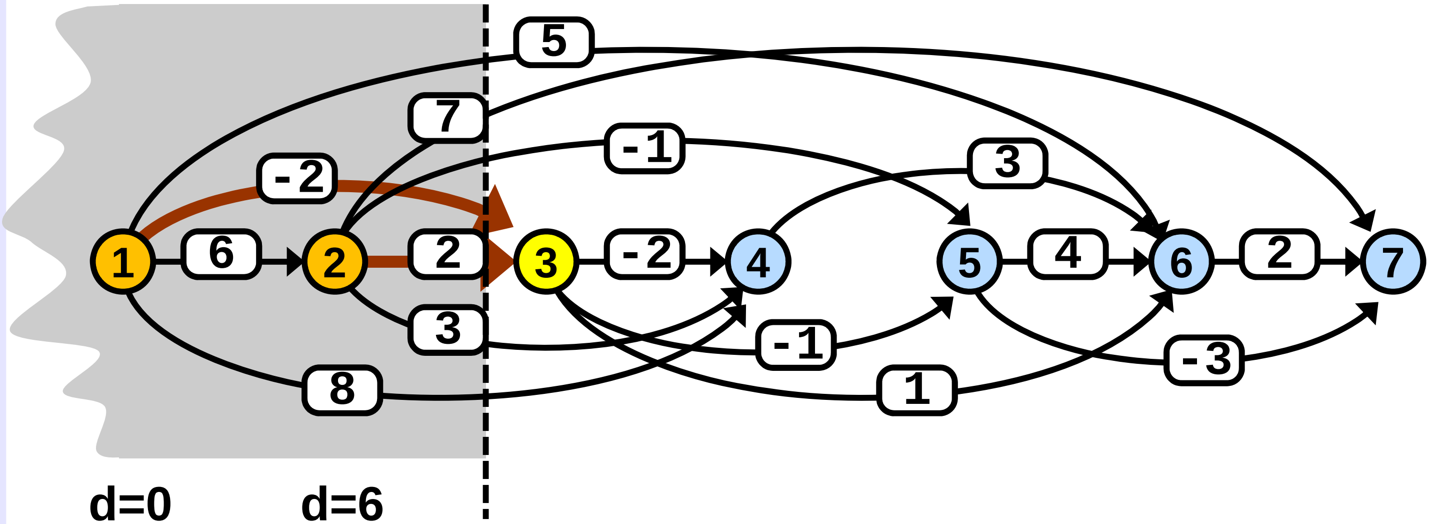
d=0

p=nil

d = max {0+6}
= 6

p=1

Longest path in DAG



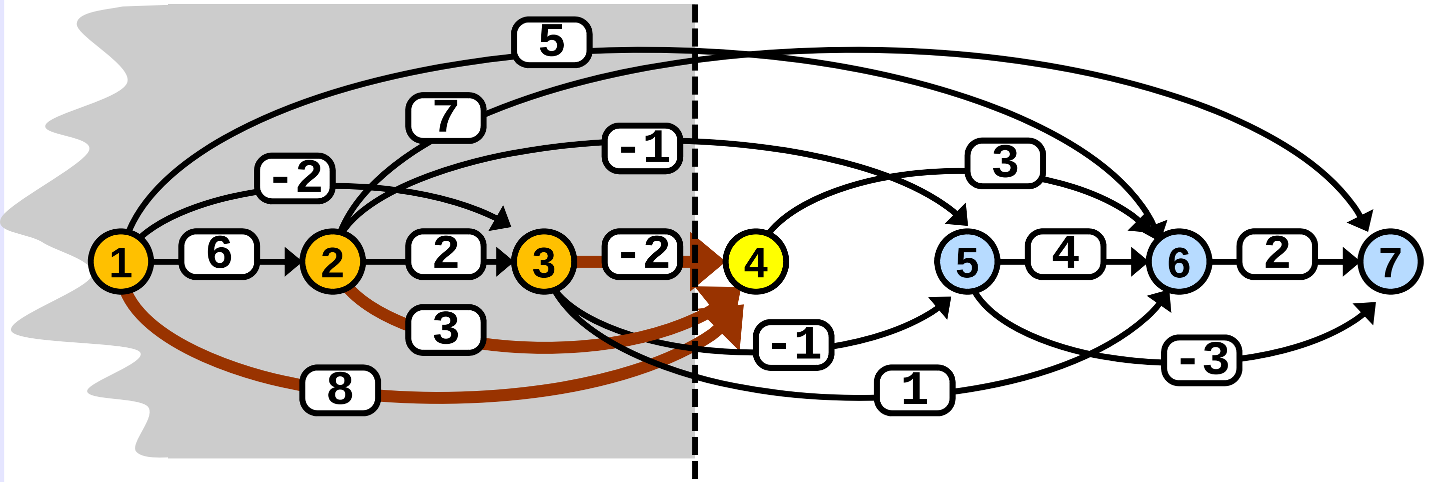
d=0
p=nil

d=6
p=1

d = max {0+-2,
6+2}
= 8

p = 2

Longest path in DAG



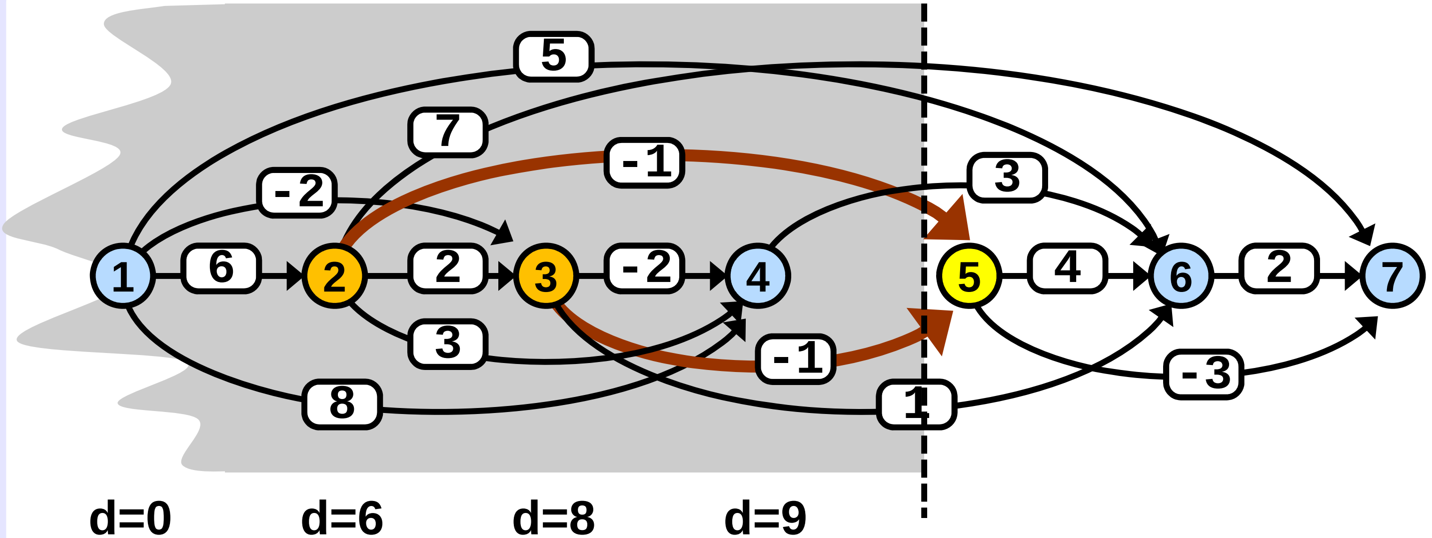
d=0 d=6 d=8

p=nil p=1 p=2

d = max {0+8,
6+3,
8+-2}
= 9

p = 2

Longest path in DAG



d=0 p=nil

d=6 p=1

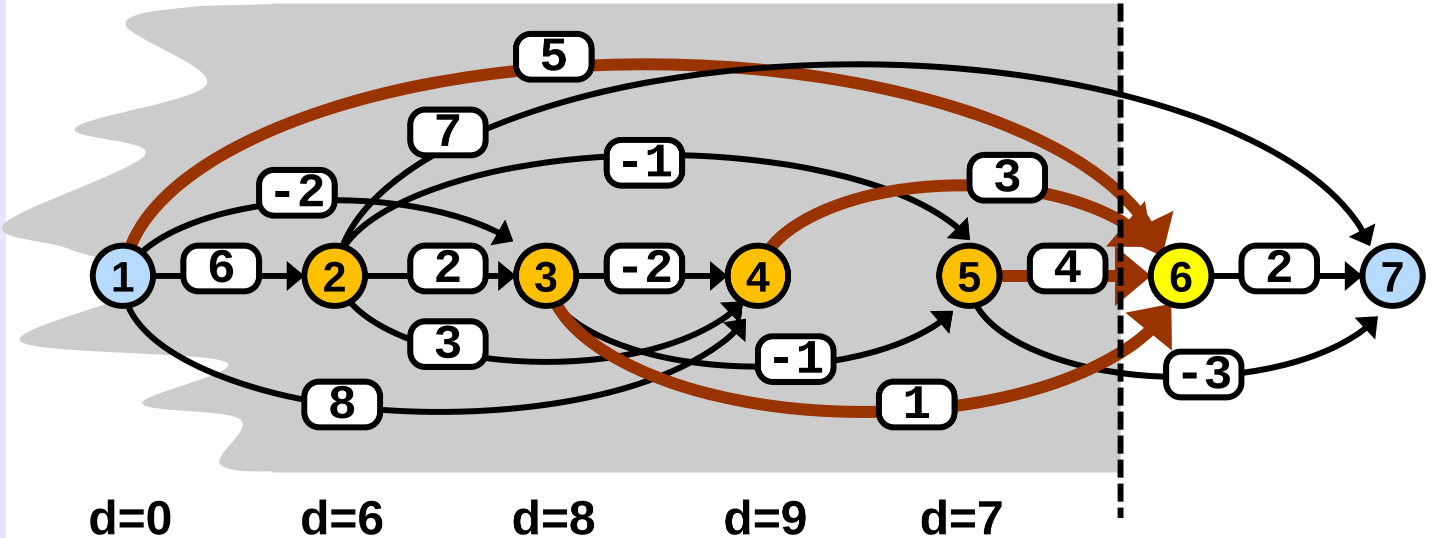
d=8 p=2

d=9 p=2

d = max {6+-1,
8+-1}
= 7

p = 3

Longest path in DAG



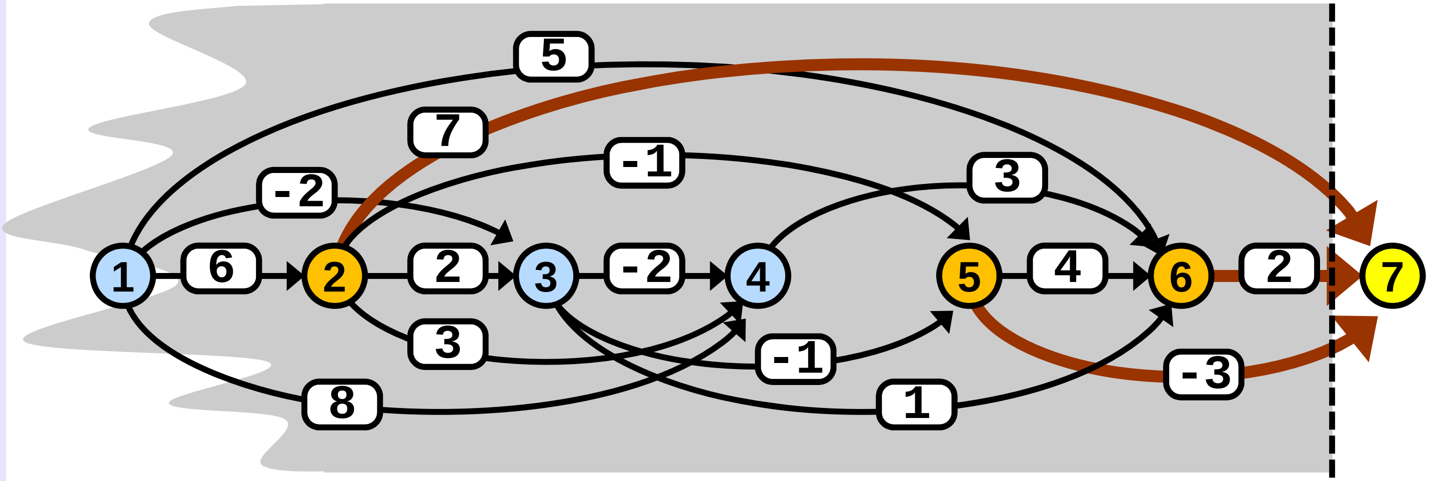
d=0 d=6 d=8 d=9 d=7

p=nil p=1 p=2 p=2 p=3

d = max {0+5,
8+1,
9+3,
7+4}
= 12

p = 4

Longest path in DAG



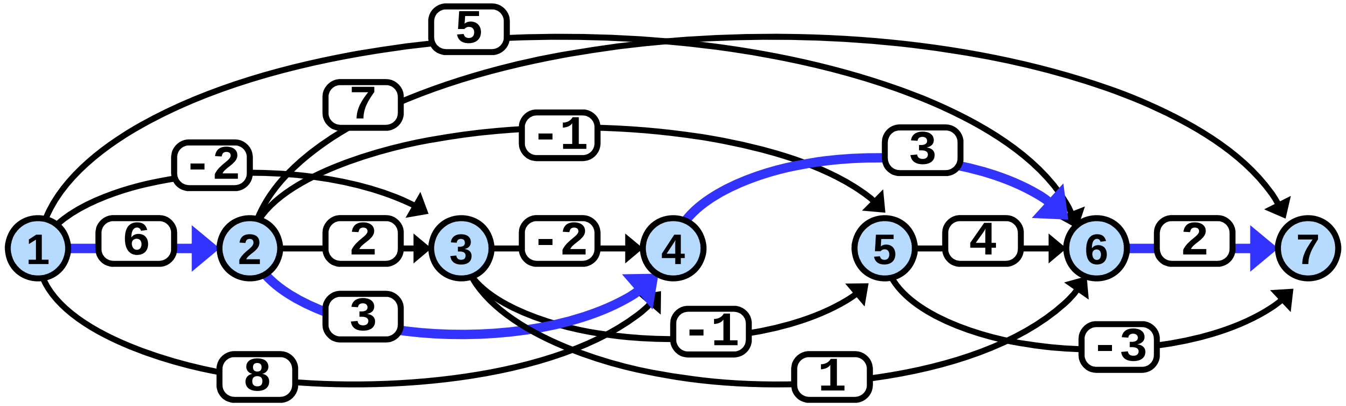
d=0	d=6	d=8	d=9	d=7	d=12
p=nil	p=1	p=2	p=2	p=3	p=4

$$d = \max\{6+7,\ 7+-3,\ 12+2\}$$

$$= 14$$

$$p = 6$$

Longest path in DAG



d=0	d=6	d=8	d=9	d=7	d=12	d=14
p=nil	p=1	p=2	p=2	p=3	p=4	p=6

Length of the longest path: 14
The longest path itself: 1 -- 2 -- 4 -- 6 -- 7

Longest path in DAG

```
0. allocate memory for distance and predecessor of each node

1. for each x in V(G):
        x.dist = negInfinity
        x.pred = null


# supposing nodes are processed
# in ascending topological order
2. for each node x in V(G):
      for each edge e = (y, x) in E(G):
         if x. dist < y.dist + e.weight:
            x. dist = y.dist + e.weight
            x.pred = y
   }
   if x. dist < 0: x.dist = 0;     # avoid negative path lengths
}
```

0. Complexity Θ(N)

1. Complexity Θ(N)

2. Complexity Θ(M),
   each edge is visited exactly once and it is processed in constant time.

Complexity: Θ(N+M)

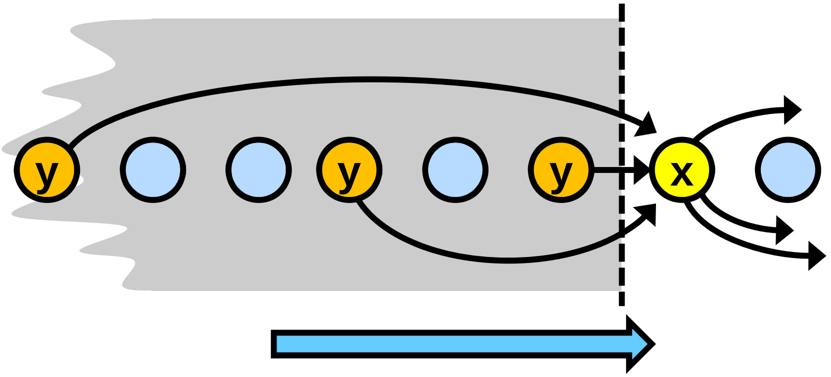
Longest path in DAG

Variant I

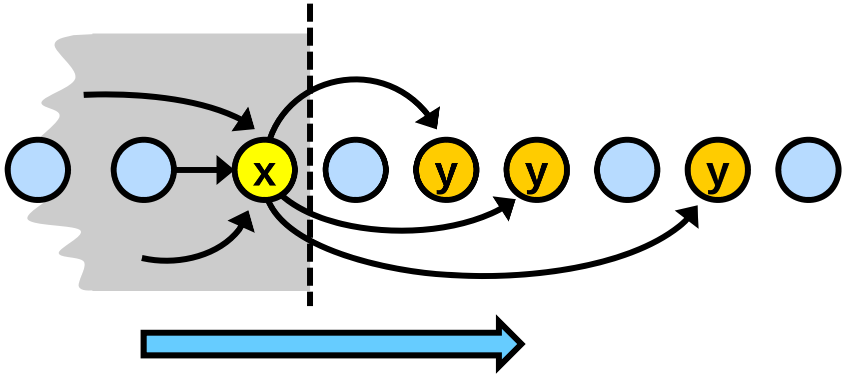
```
2. for each node x in V(G):

      for each edge e = (y, x) in E(G):
          if  x.dist < y.dist + e.weight:
              x.dist = y.dist + e.weight
              x.pred = y

      if x. dist < 0:
          x.dist = 0
```

Variant II

```
2. for each node x in V(G):

      if x. dist < 0:
          x.dist = 0

      for each edge e = (x, y) in E(G):
          if y.dist < x.dist + e.weight:
              y.dist = x.dist + e.weight
              y.pred = x
```



order of processing
= topological order



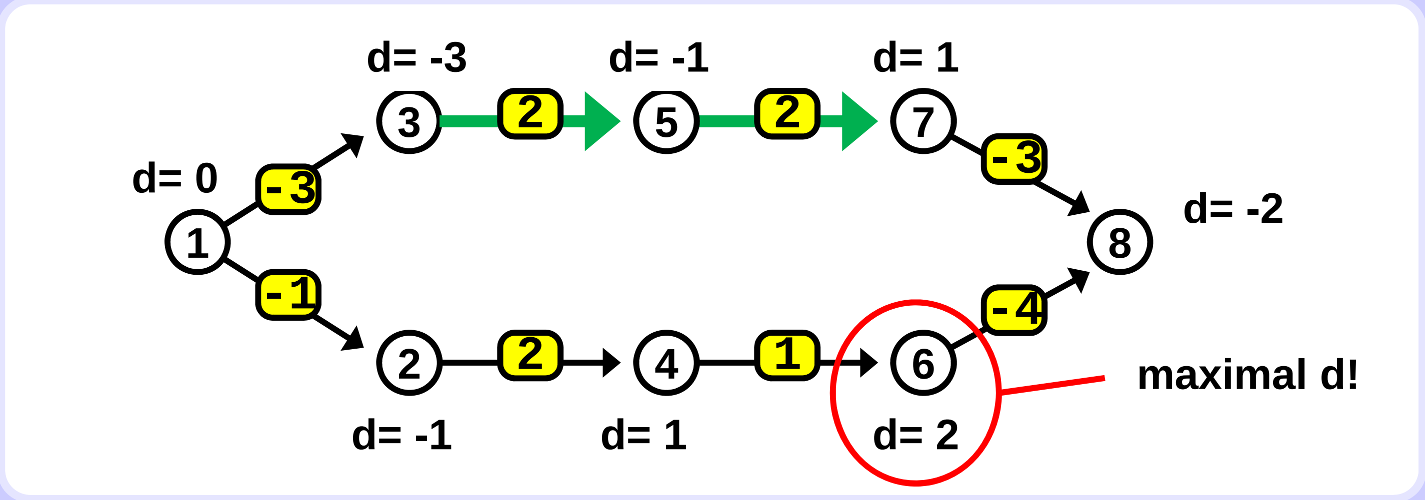
order of processing
= topological order

Longest path in DAG

Warning

Algorithms presented in the literature and on the web often solve the maximum path in DAG problem only for non-negative edge weights and do not mention explicitely this limitation.
Those algorithms cannot handle DAG containing negative weight edges.

Incorrect result produced by algorithm
expecting only non-negative edge weights

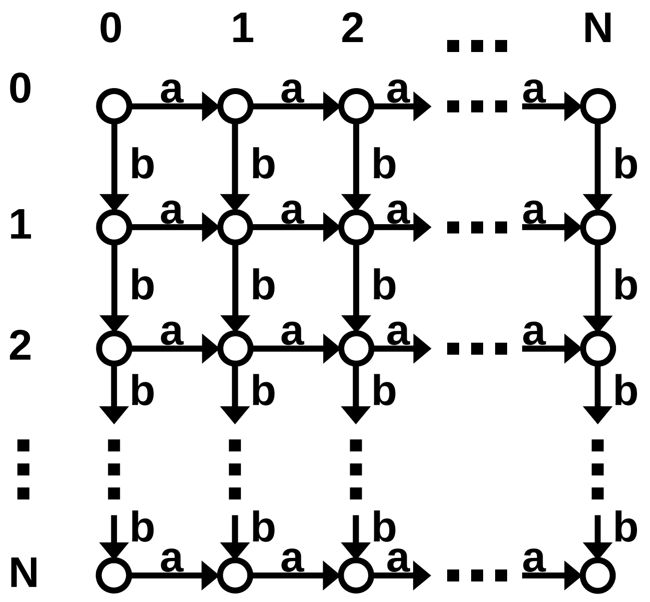


Actual maximum path is 3 -- 5 -- 7 which weight is 4.
Algorithm limited to non-negative weights finds
only suboptimal path 1 -- 2 -- 4 -- 6 which weight is 2.

Longest path in DAG

Problem of reconstucting all optimal paths
-- the number of paths can be too big.

Example



Each path from the root to the leaf is optimal, its weight is N·(a+b).

Number of all paths is Comb(2N, N), and it holds 2^N < Comb(2N, N) < 4^N.

The numbert of optimal paths thus grows exponentially with the value of N.

N	# of optimal paths
1	2
10	184756
20	137846528820
30	118264581564861424
40	107507208733336176461620