COMP251: Dynamic programming (2)

Jérôme Waldispühl & Roman Sarrazin-Gendron
School of Computer Science
McGill University

Based on (Kleinberg & Tardos, 2005) & Slides by K. Wayne

Outline

- Dynamic programming in a nutshell
- Single source shortest path with Bellman-Ford
- The Knapsack problem!

Dynamic Programming in a nutshell

- Optimization method introduced by Richard Bellman in 1953
- Principle of Optimality: "*An optimal policy has the property that what-ever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision*" (Bellman, 1957)
- The term "dynamic programming" was chosen by Bellman to emphasize the progressive approach to solving an optimization problem (and because it is impressive!).
- The technique aims to solve the problem by increasing size (i.e., once you have the optimal solution for all sub-problems of size 1, you can compute solution for size 2 using previously computed results, etc.)
- We say we use a **bottom-up** approach
- Define a data structure that stores the value to optimize on subproblems of given size and iterate over increasing sizes.

What is Richard Bellman also known for?

SINGLE SOURCE SHORTEST PATHS

Modeling as graphs

Input:

- Directed graph G = (V, E)
- Weight function w : E → R

Weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$= \sum_{k=1}^{n} w(v_{k-1}, v_k)$$

= sum of edge weights on path *p*.

Shortest-path weight *u* to *v*:

$$\delta(u,v) = \begin{cases} \min\left\{w(p) : u \overset{p}{\mapsto} v\right\} & \text{If there exists a path u} \rightsquigarrow \text{v.} \\ \infty & \text{Otherwise.} \end{cases}$$

Shortest path *u* to *v* is any path *p* such that $w(p) = \delta(u,v)$.

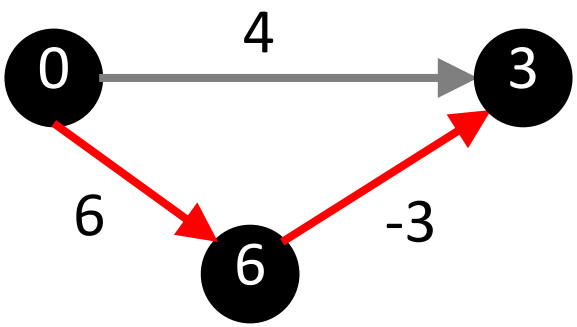
Generalization of breadth-first search to weighted graphs.

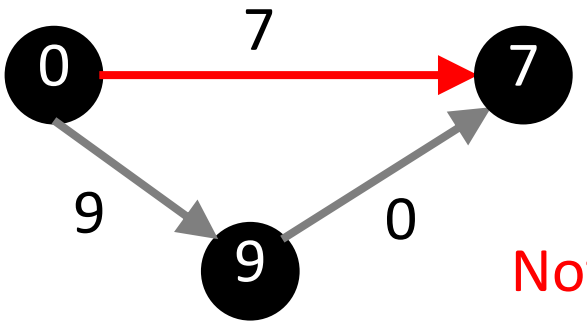
Dijkstra's algorithm

- No negative-weight edges.
- Weighted version of BFS:
  - Instead of a FIFO queue, uses a **priority queue**.
  - Keys are shortest-path weights (*d*[v]).
- Greedy choice: At each step we choose the light edge.

Can we adapt Dijkstra's to deal with negative weight edges?

- Allow re-insertion in queue? $\Longrightarrow$ Exponential running time…
- Add constant to each edge?





Not working…

Bellman-Ford Algorithm

If there is a negative cycle, there is no point of computing the shortest path. We only want to decide in finite time if such cycle exists.

- Allows negative-weight edges.
- Computes $d[v]$ and $\pi[v]$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

If Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

By contrast, Dijkstra's finishes on a graph with negative weight edges, but **it does not detect the occurrence of them** and thus the correctness of the output.

Bellman-Ford Algorithm

- Can have negative-weight edges.
- Will "detect" **reachable** negative-weight cycles.

We introduce the algorithm first. We will see later how to present it as a dynamic programming algorithm.

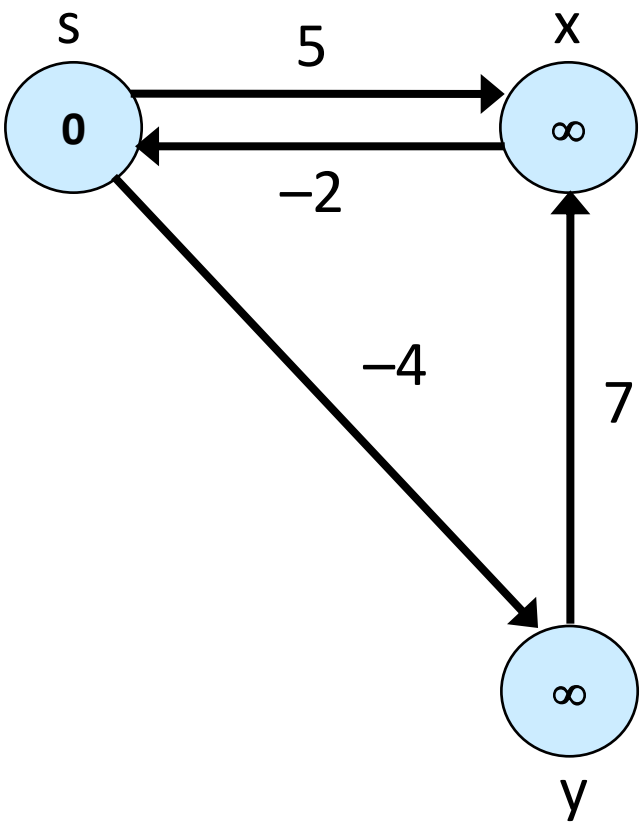
```
Initialize(G, s);
for i := 1 to |V[G]| –1 do
      for each (u, v) in E[G] do
            Relax(u, v, w)
for each (u, v) in E[G] do
      if d[v] > d[u] + w(u, v) then
            return false
return true
```

Time Complexity is O(VE).

Why?

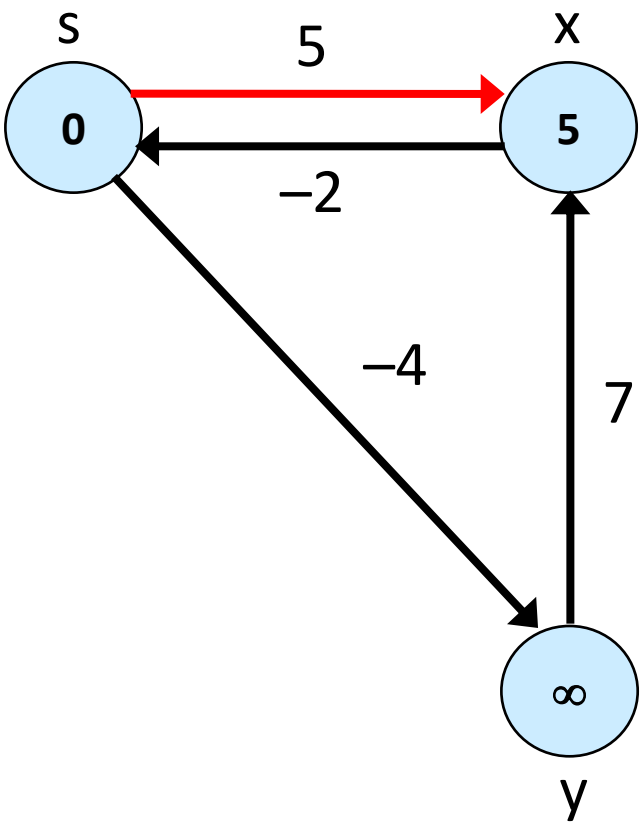
Note: We no longer use a priority queue. We relax all edges at each iteration.

Example



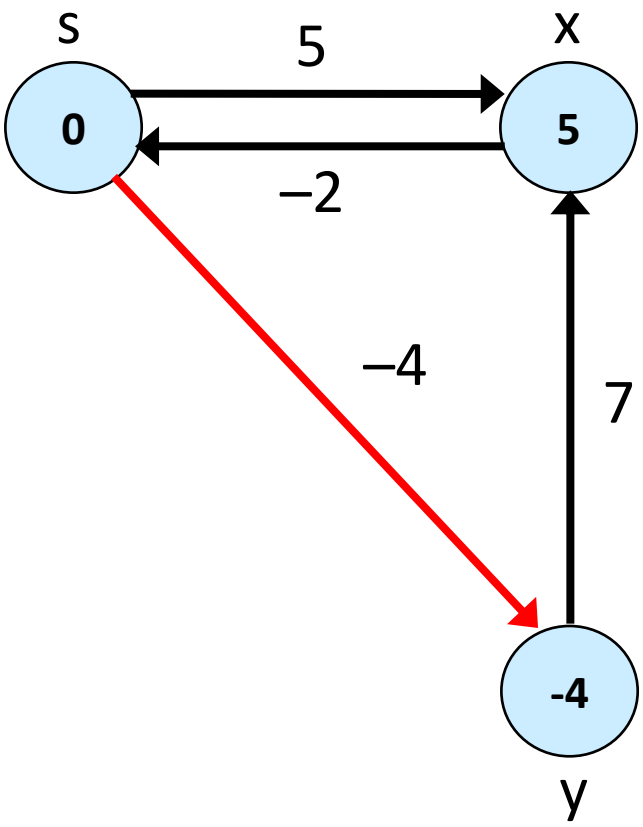
Example

Iteration 1



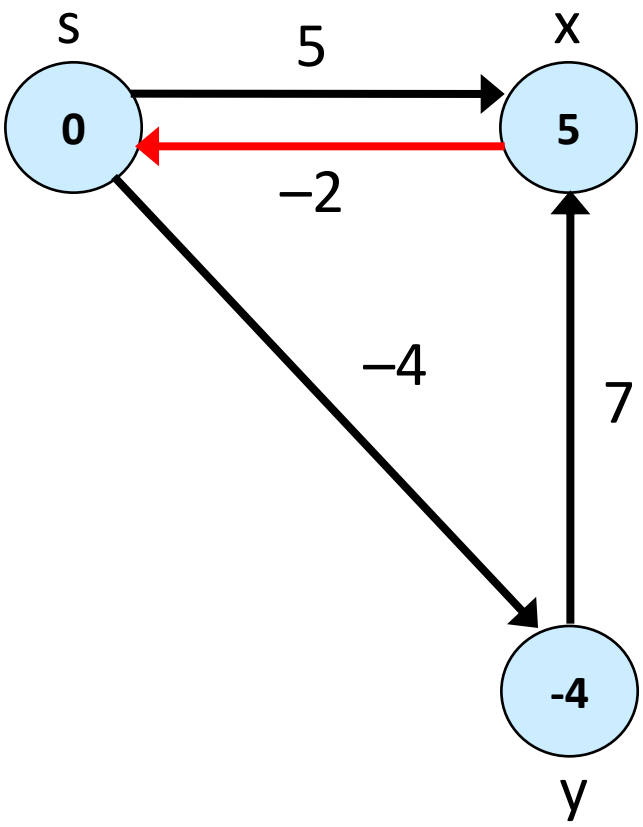
Example

Iteration 1



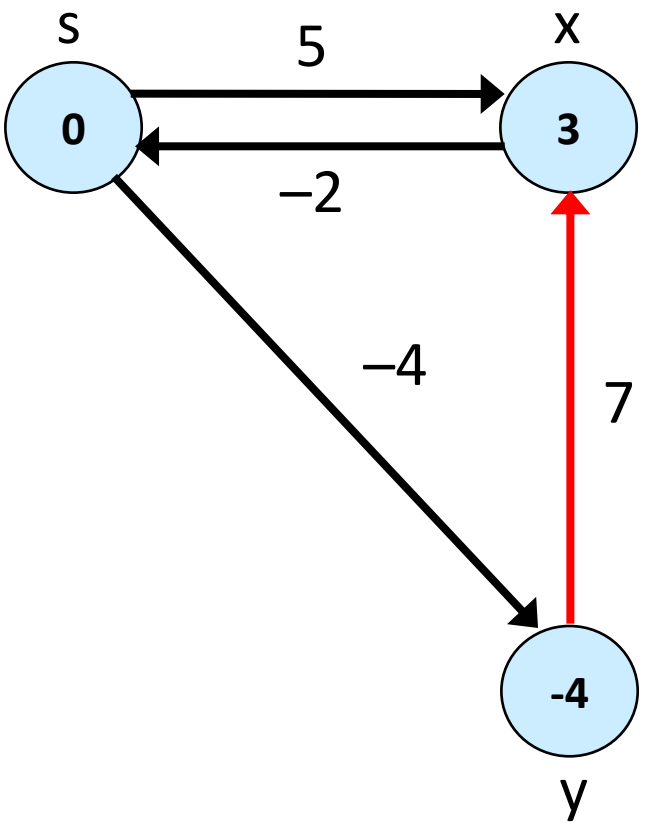
Example

Iteration 1



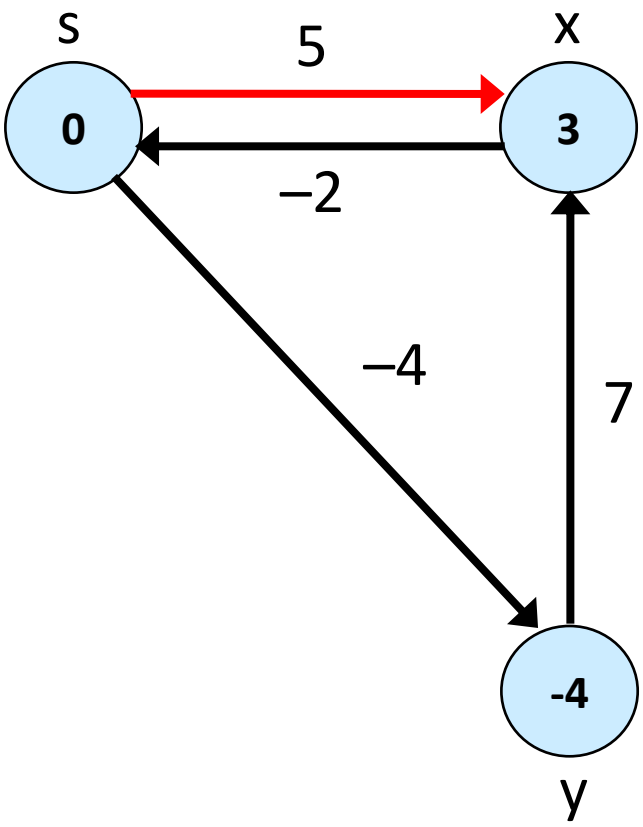
Example

Iteration 1



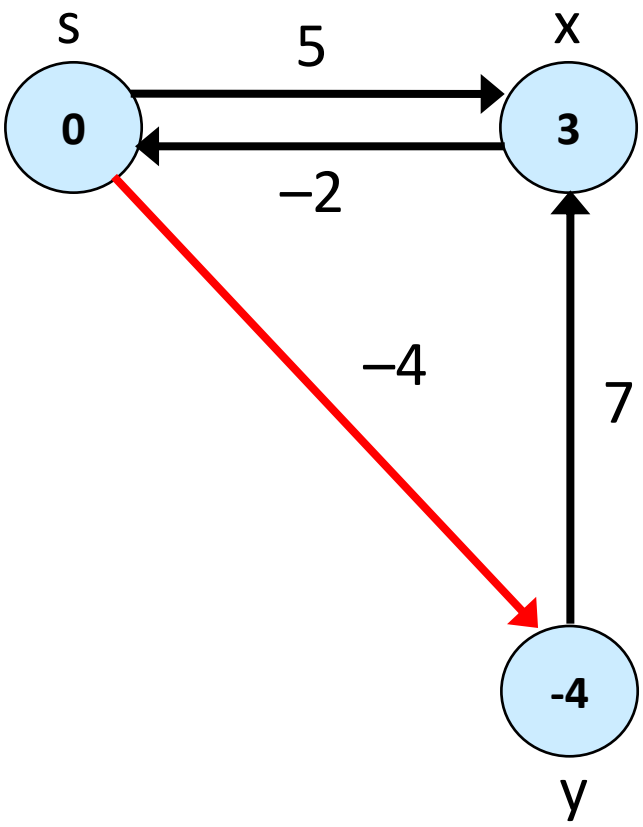
Example

Iteration 2



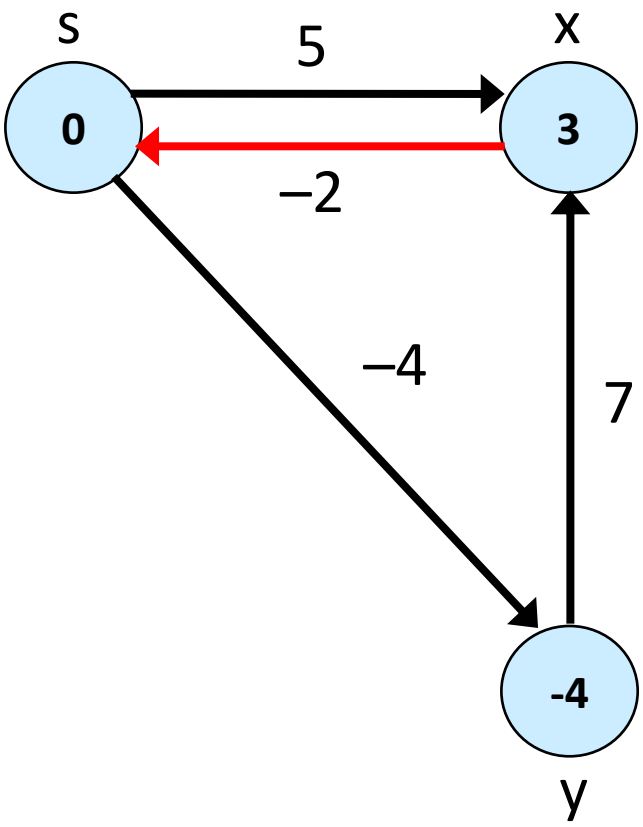
Example

Iteration 2



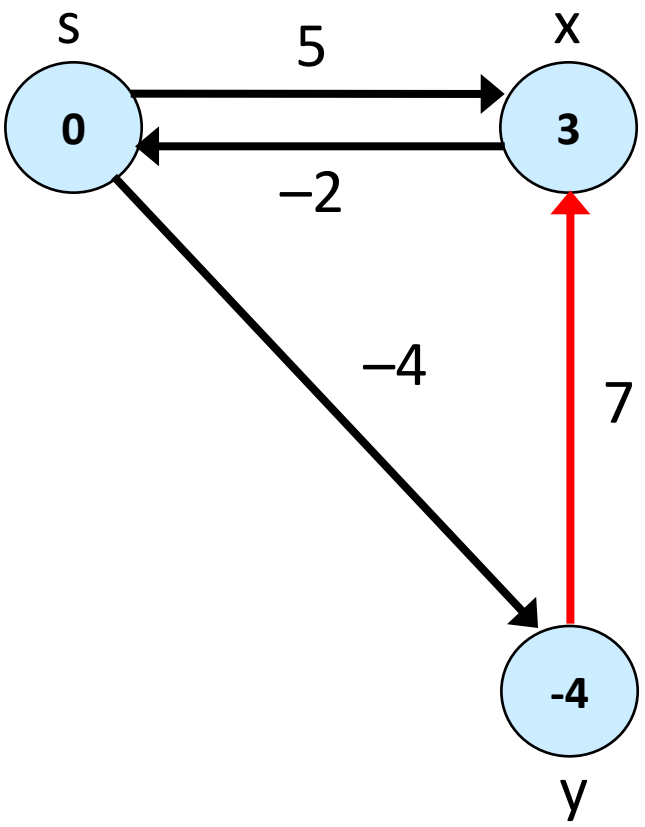
Example

Iteration 2



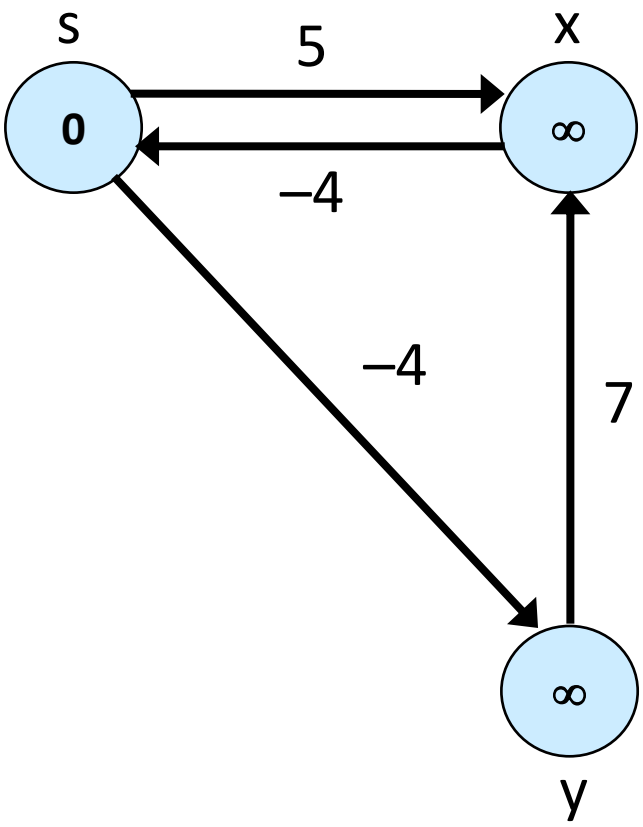
Example

Iteration 2



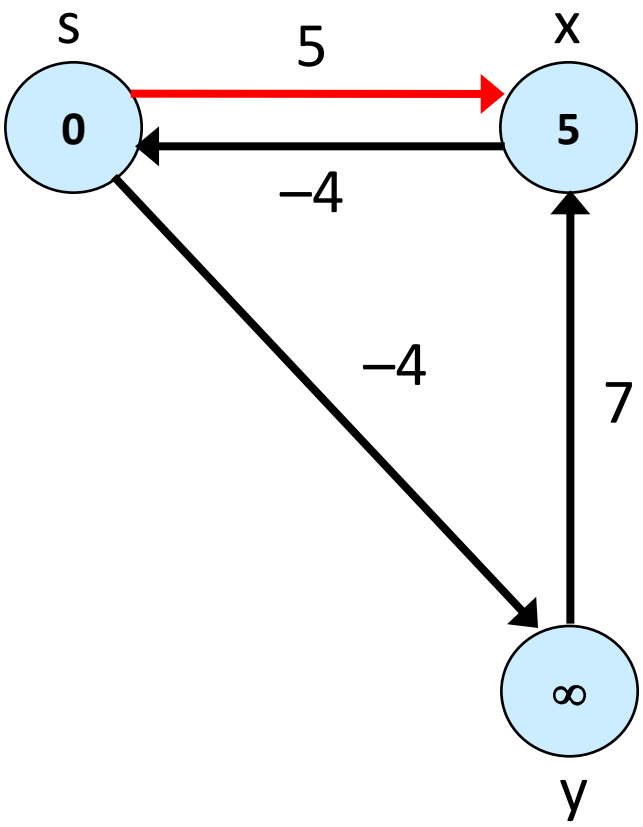
Bellman-Ford terminates and all shortest paths are computed.

Example 2



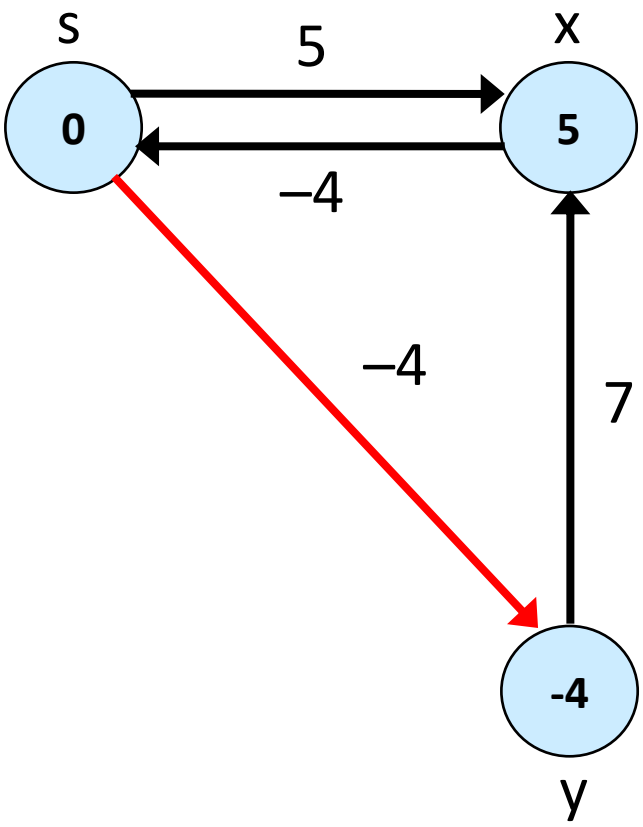
Example 2

Iteration 1



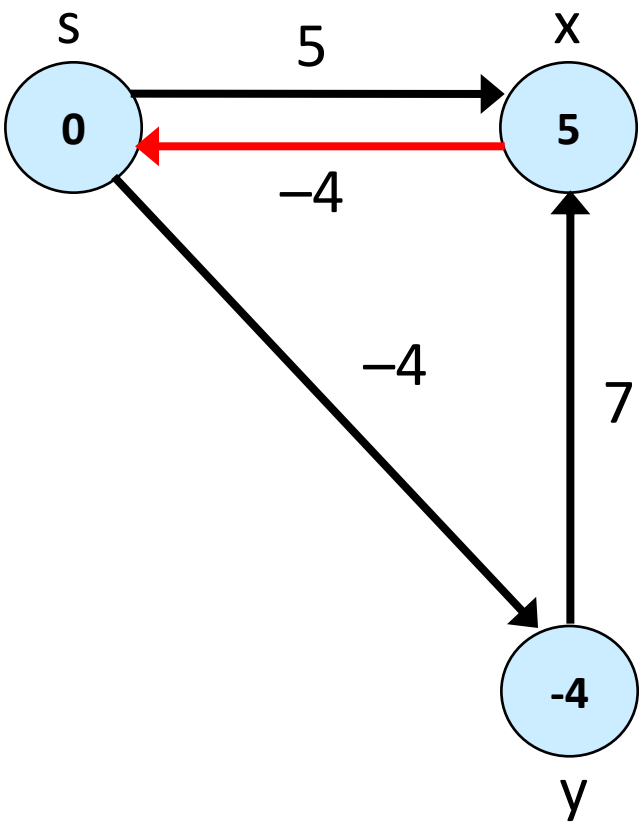
Example 2

Iteration 1



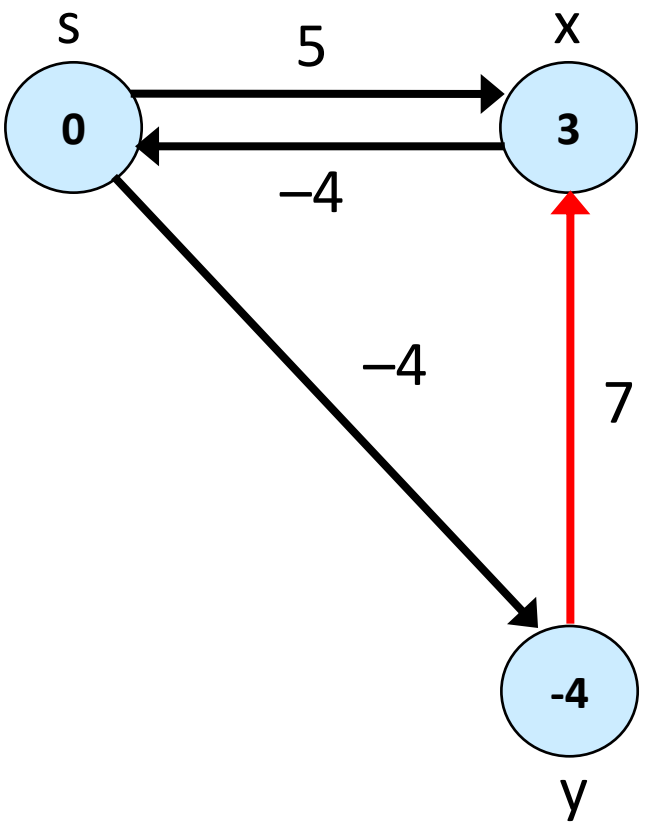
Example 2

Iteration 1



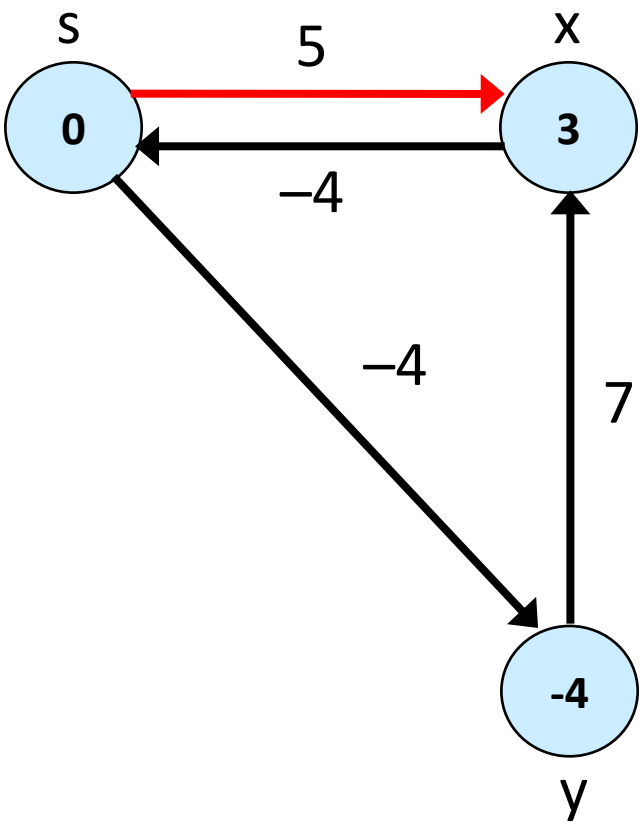
Example 2

Iteration 1



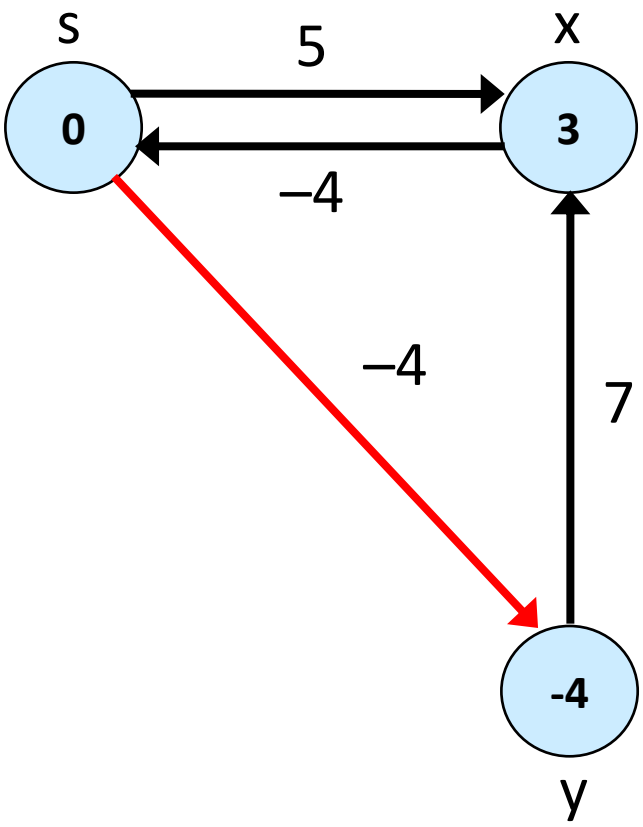
Example 2

Iteration 2



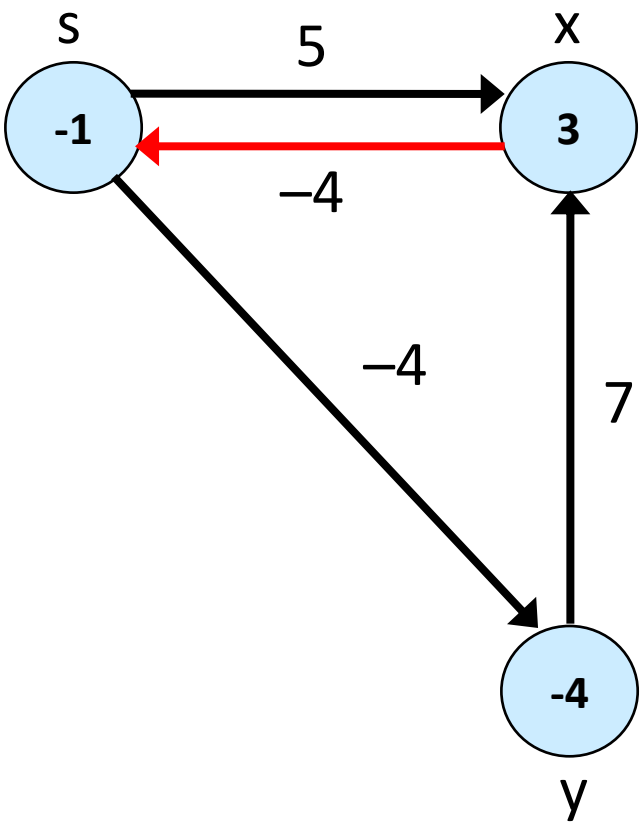
Example 2

Iteration 2



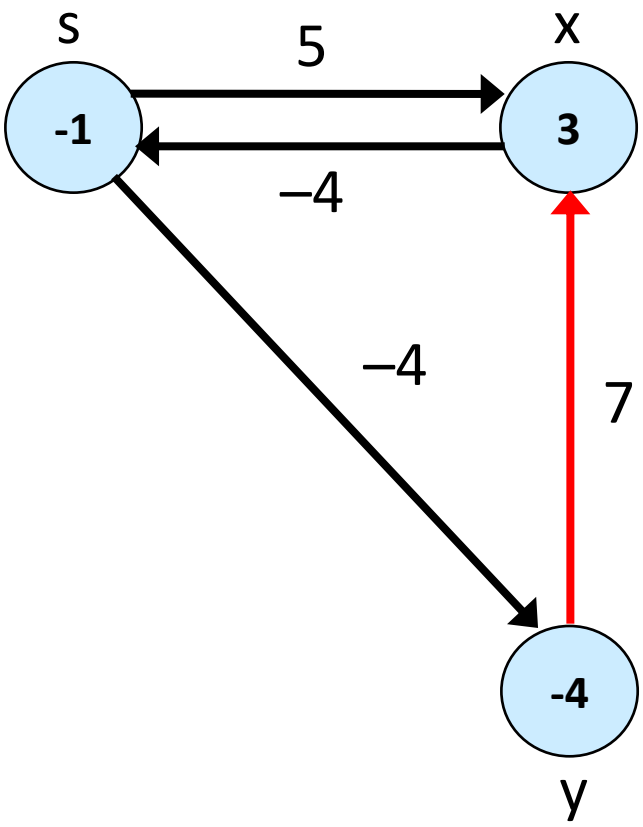
Example 2

Iteration 2



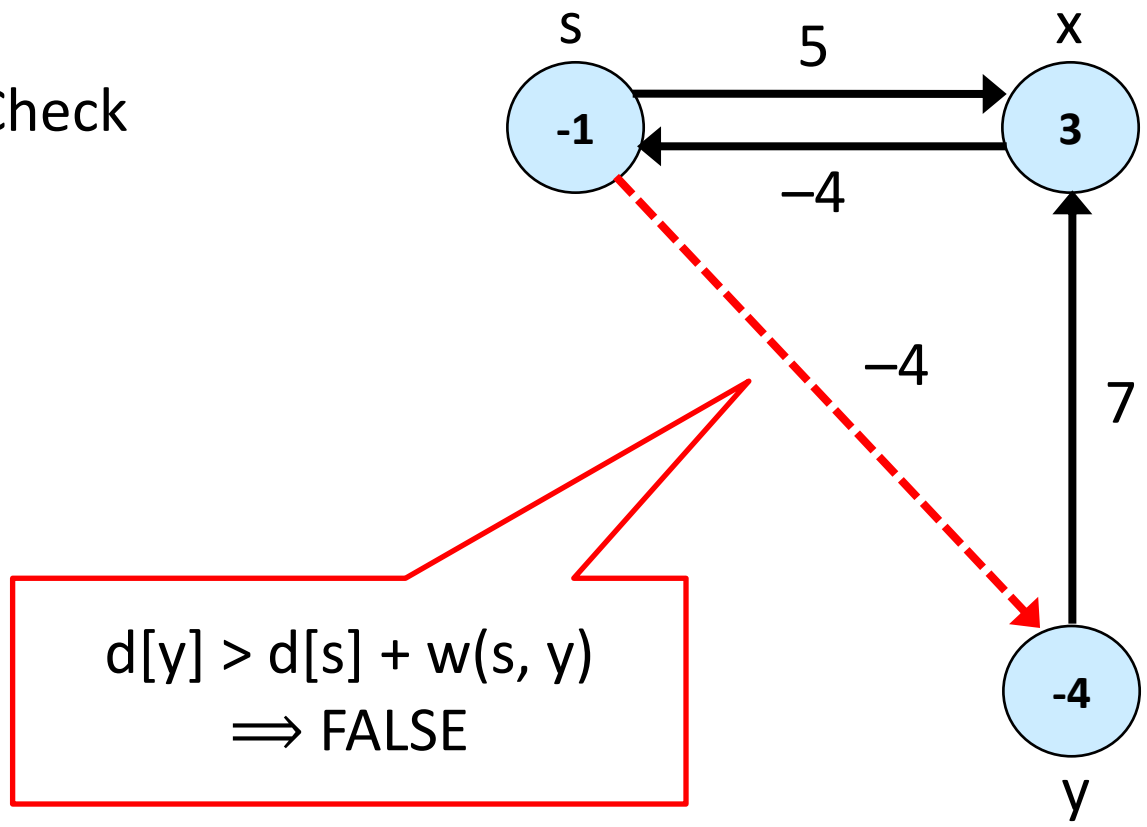
Example 2

Iteration 2



Example 2

Check



At the end, all vertices have been relaxed on any order possible. If a shortest path estimate still decreases after relaxing another edge and visiting a vertex twice, it implies that there is a negative weight cyle.

Another Look at Bellman-Ford

Bellman-Ford is essentially a dynamic programming algorithm.

Let d(i, j) = cost of the shortest path from s to i that is at most j hops.

$$d(i,j) = \begin{cases} 0 & \textbf{if } i = s \wedge j = 0 \\ \infty & \textbf{if } i \neq s \wedge j = 0 \\ \min(\{d(k, j-1) + w(k, i) : i \in Adj(k)\} \cup \{d(i, j-1)\}) & \textbf{if } j > 0 \end{cases}$$

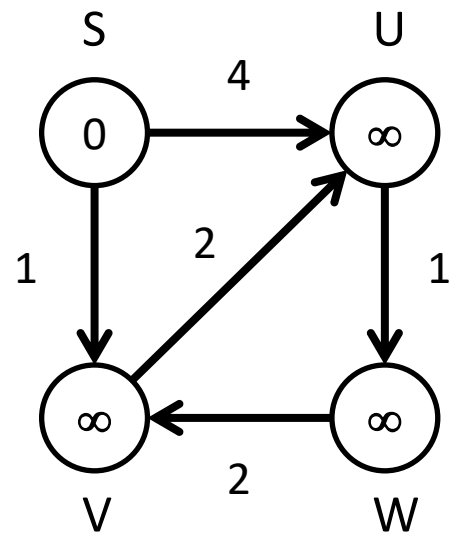
d(*,*) is our dynamic array!

We fill the array d in increasing length of the path (i.e., #hops).

	i →	z	u	v	x	y
		1	2	3	4	5
j ↓	0	0	∞	∞	∞	∞
	1	0	6	∞	7	∞
	2	0	6	4	7	2
	3	0	2	4	7	2
	4	0	2	4	7	–2

The third row of the recursive formula is the relaxation of all in-going edges to vertex i.

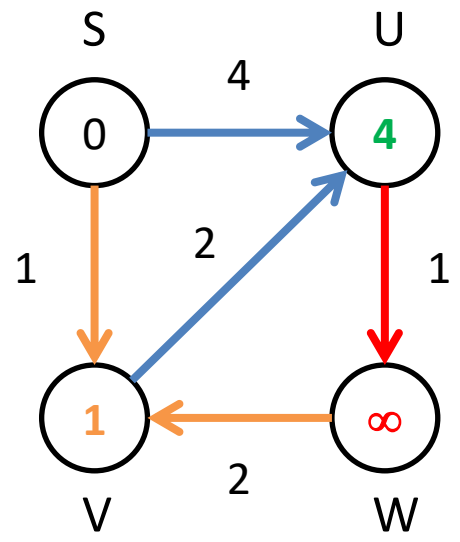
Example



	S	U	V	W
0	0	∞	∞	∞
1				
2				
3				
4				

Initialization. The first row stores the distance from the source to all vertex in the graph.

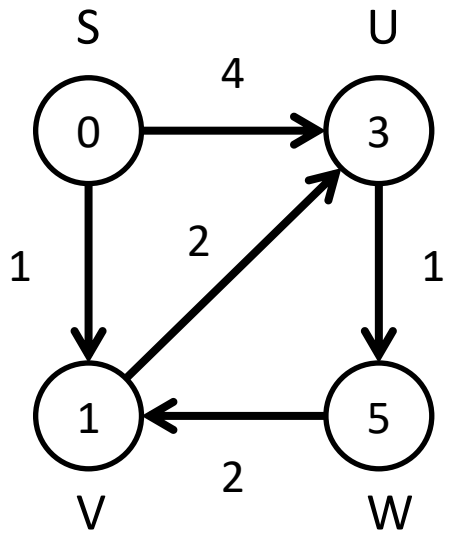
Example



	S	U	V	W
0	0	∞	∞	∞
1	0	4	1	∞
2				
3				
4				

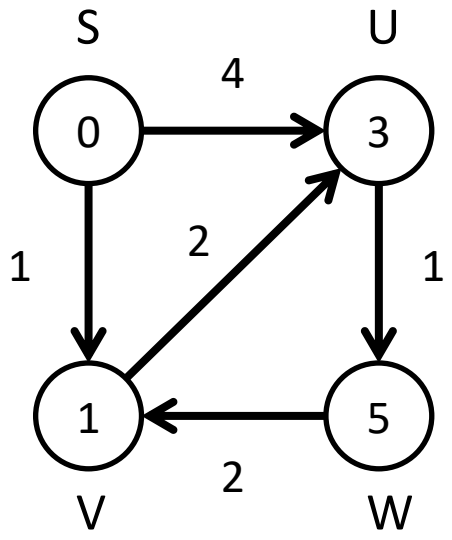
We only need/consider values stored in the row above the current one that is being filled.

Example



	S	**U**	**V**	**W**
0	0	∞	∞	∞
1	0	4	1	∞
2	0	3	1	5
3				
4				

Example

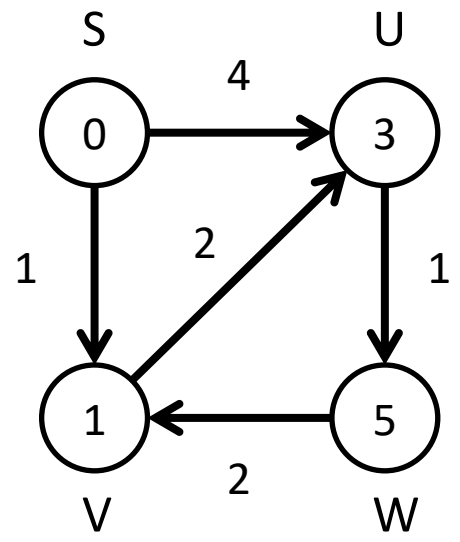


	S	U	V	W
0	0	∞	∞	∞
1	0	4	1	∞
2	0	3	1	5
3	0	3	1	4
4				

At this point, we considered all paths with |V|-1 edges, which means we considered any possible path including those visiting all vertices.

Are we done?

Example

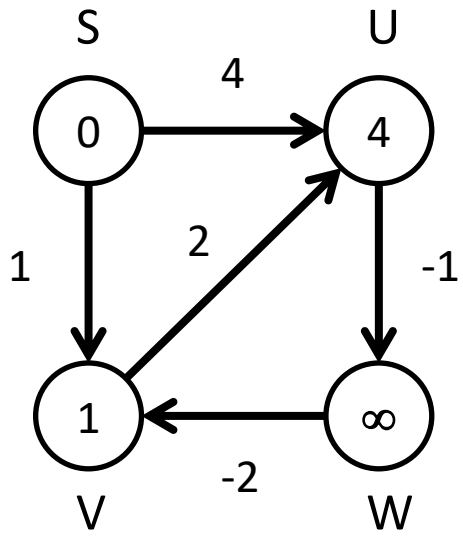


	S	U	V	W
0	0	∞	∞	∞
1	0	4	1	∞
2	0	3	1	5
3	0	3	1	4
4	0	3	1	4

We relax all edges one more time (i.e., we fill another row) to check the result is stable!

If the two last rows are identical, there is no negative weight cycle, and your result is stored in the last row!

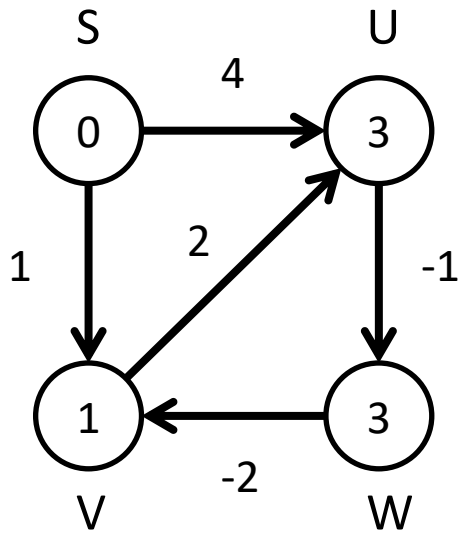
Example 2



	S	U	V	W
0	0	∞	∞	∞
1	0	4	1	∞
2				
3				
4				

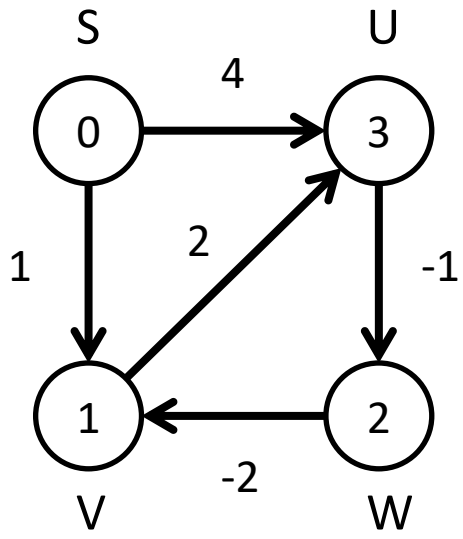
Let's see what is happening when there is a negative cycle…

Example 2



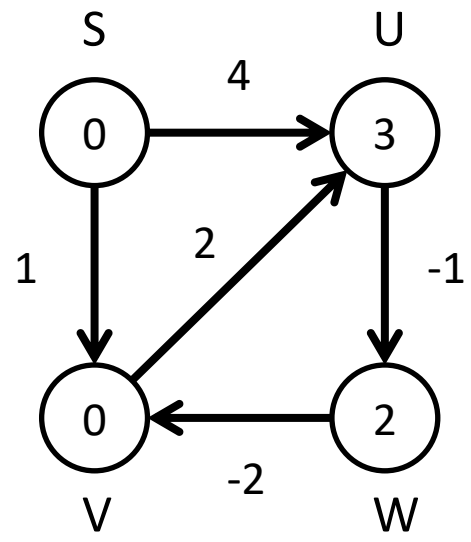
	S	U	V	W
0	0	∞	∞	∞
1	0	4	1	∞
2	0	3	1	3
3				
4				

Example 2



	S	U	V	W
0	0	∞	∞	∞
1	0	4	1	∞
2	0	3	1	3
3	0	3	1	2
4				

Example 2



	S	U	V	W
0	0	∞	∞	∞
1	0	4	1	∞
2	0	3	1	3
3	0	3	1	2
4	0	3	0	2

The last two rows are different. There must be a negative weight cycle! Return False.

Note: You do not need to wait to observe a discrepancy for all vertices on the negative weight cycle. You only need to guarantee you observe a decrease of the shortest path estimate for at least one vertex.

Pseudo-Code

```
Input: weighted directed graph G(V,E,w)
Output: Boolean indicating the absence of a negative
weight cycle

for i in V do
   d(i,0)=∞
d(s,0)=0
for j=1 to |V|-1 do
   for i in V do
      d(i,j)=d(i,j-1)
      for (k,i) in E do
         if d(k,j-1)+w(k,i)<d(i,j) then
             d(i,j)= d(k,j-1)+w(k,i)
for i in V do
   if d(i,|V|)!= d(i,|V|-1) then
      return False
return True
```

Correctness of the algorithm

You need to prove that your formulation of the problem satisfies the optimal substructure property.

In this case, we simply need the lemma introduced in lecture 13

Lemma (shortest path's optimal substructure)
Any subpath of a shortest path is a shortest path.

Proof : See lecture 13

Then, for Bellman-Ford, all you need to show is that you can compute the shortest path (from s to i) of length j using only the knowledge of shortest path of length j-1.

You can derive this result from the optimal substructure property.

Exercise!

KNAPSACK PROBLEM

Knapsack problem

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.

Ex. $\{ 1, 2, 5 \}$ has value 35.

Ex. $\{ 3, 4 \}$ has value 40.

Ex. $\{ 3, 5 \}$ has value 46 (but exceeds weight limit).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance (weight limit W = 11)

Greedy by value. Repeatedly add item with maximum v_i.

Greedy by weight. Repeatedly add item with minimum w_i.

Greedy by ratio. Repeatedly add item with maximum ratio v_i / w_i.

Observation. None of greedy algorithms is optimal.

(slide by K. Wayne)

False start…

Let's try a similar approach to those used for the weighted scheduling problem:

Def. $OPT(i)$ = max profit subset of items $1, \ldots, i$.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$.

optimal substructure property
(proof via exchange argument)

Case 2. OPT selects item i.

- Selecting item i does not immediately imply that we will have to reject other items.
- Without knowing what other items were selected before i, we don't even know if we have enough room for i.

i.e., After selecting the best solution out of { 1, 2, … , i-1 }, we do not know if we can add I without exceeding the weight limit.

Conclusion. Need more subproblems!

(slide by K. Wayne)

New variable

Def. $OPT(i, w)$ = max profit subset of items $1, \ldots, i$ with weight limit w.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using weight limit w.

Case 2. OPT selects item i.

- New weight limit $= w - w_i$.
- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using this new weight limit.

optimal substructure property
(proof via exchange argument)

The new variable enable us to decide if there is enough room to accommodate item i.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

(slide by K. Wayne)

Dynamic programming algorithm

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

$M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

FOR $w = 1$ TO W

IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w - w_i] \}.$

We are filling the dynamic array by increasing number of items and then weight limit.

RETURN $M[n, W]$.

Example

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Max weight W = 11

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

w →

i ↓

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0											
{1,2}	0											
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0											
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1										
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{	0	1	6									
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

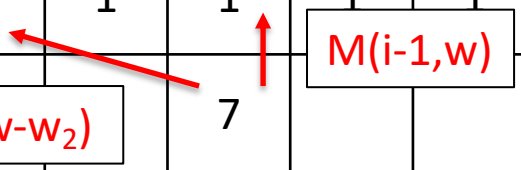
$V_2+M(i-1,w-w_2)$

M(i-1,w)

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}				7								
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											



i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
{1,2,3,4}	0											
{1,2,3,4,5}	0											

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
{1,2,3,4,5}	0											

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1					7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19					25
{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
{1,2,3,4,5}	0	1	6	7	7	18	22	28	29	34	35	40

Item 3 in solution

Item 4 in solution

Analysis

Theorem. There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(n\,W)$ time and $\Theta(n\,W)$ space.

Pf.

weights are integers between 1 and W

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\,W)$ table entries.
- After computing optimal values, can trace back to find solution: take item i in $OPT(i, w)$ iff $M[i, w] < M[i-1, w]$. ▪

Remarks.

- Not polynomial in input size! ← "pseudo-polynomial"
- Decision version of knapsack problem is NP-COMPLETE. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]

(slide by K. Wayne)