COMP251: Dynamic programming (2)

Jérôme Waldispühl

School of Computer Science

McGill University

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

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***:**

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

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

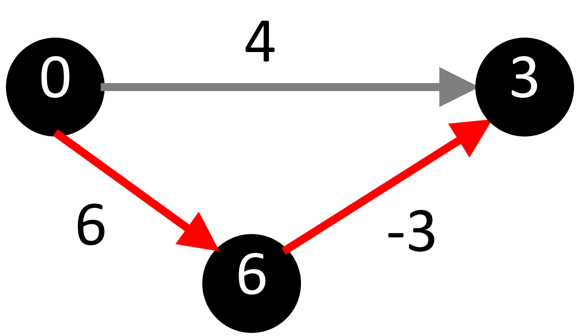
Generalization of breadth-first search to weighted graphs.

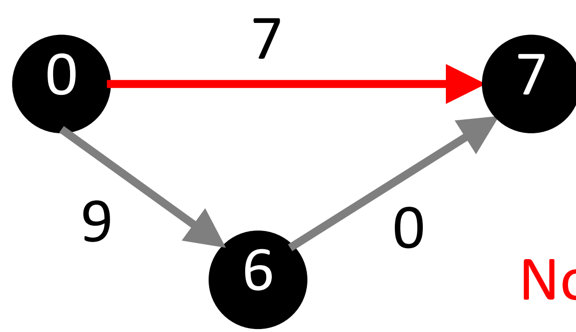
Dijkstra’s algorithm

- 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.

How 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

- 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.

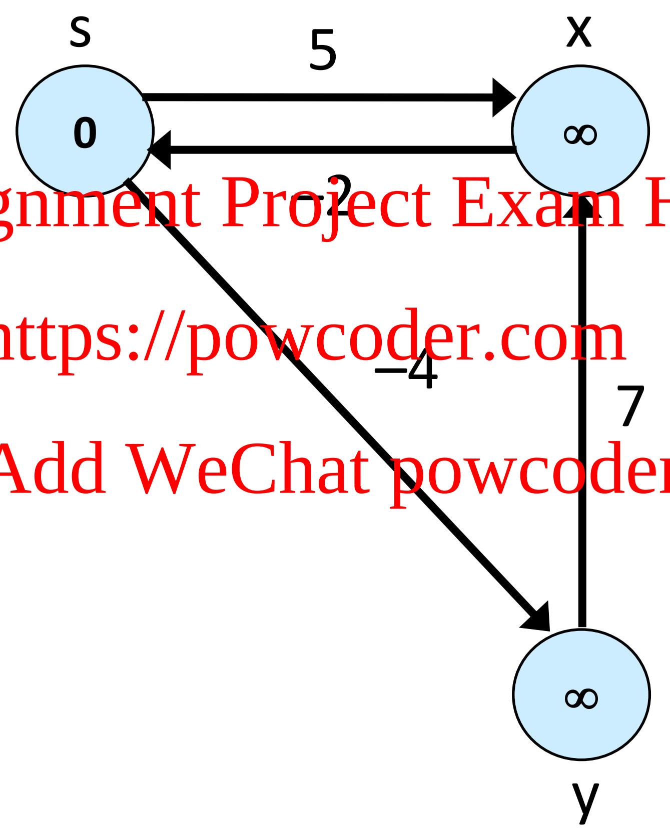
Bellman-Ford Algorithm

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


```
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
```

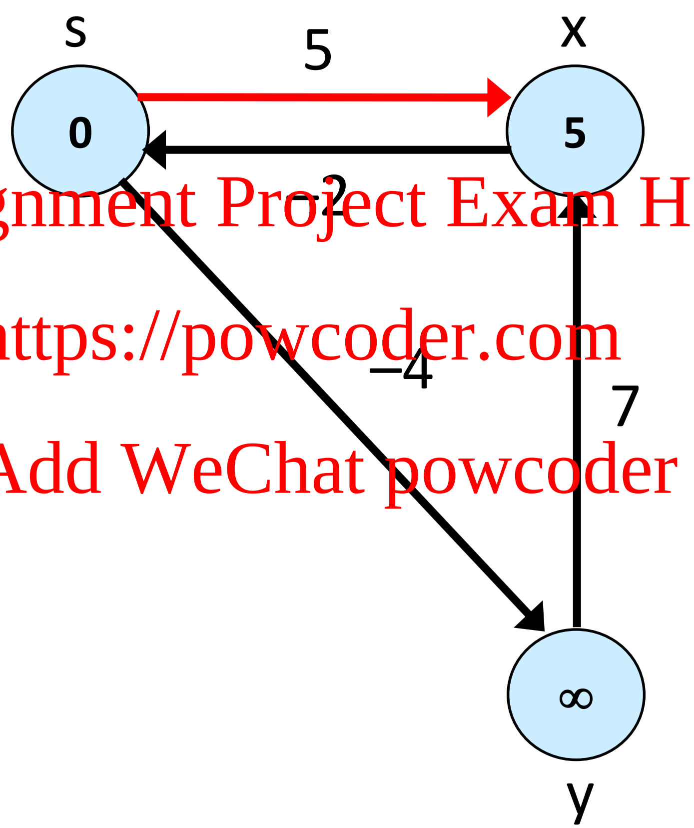

Time Complexity is O(VE).

Example



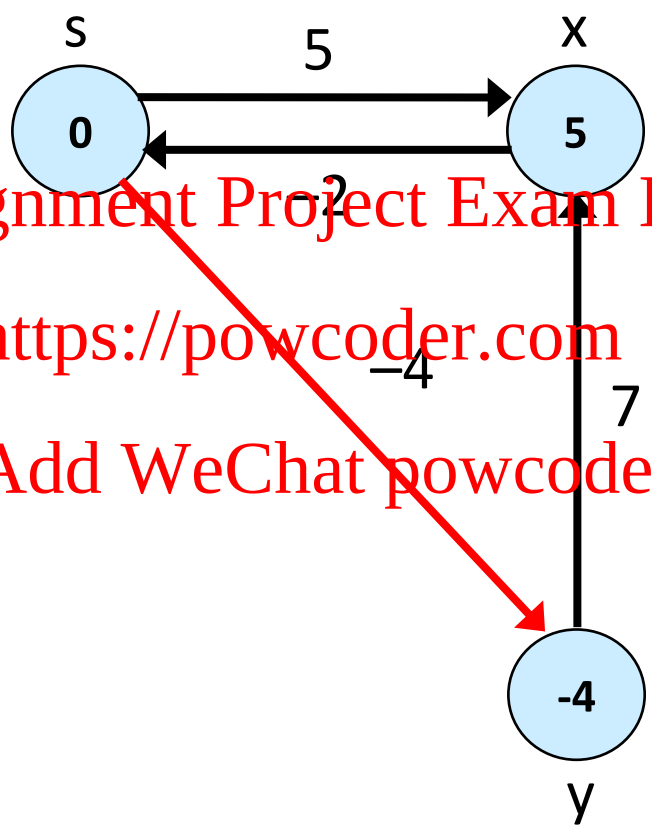
Example

Iteration 1



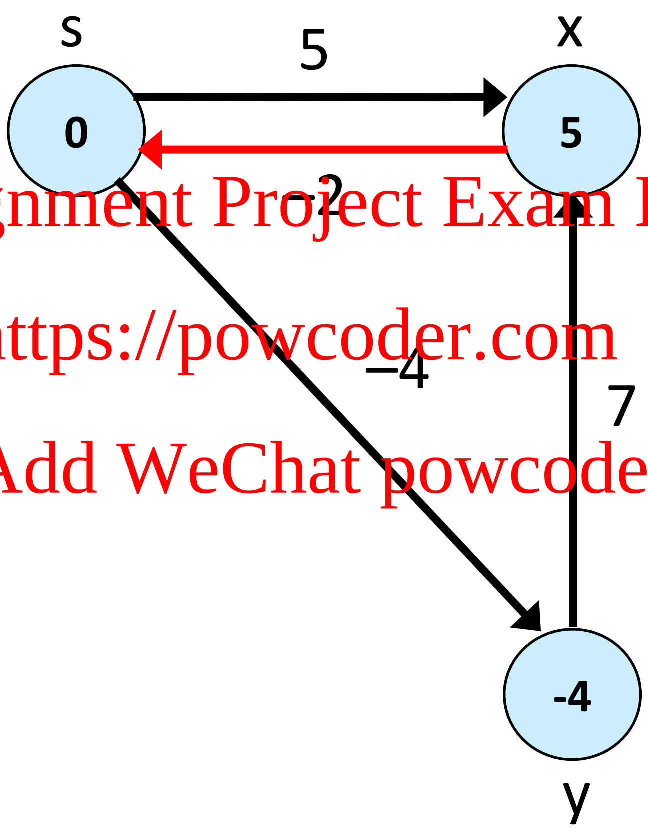
Example

Iteration 1



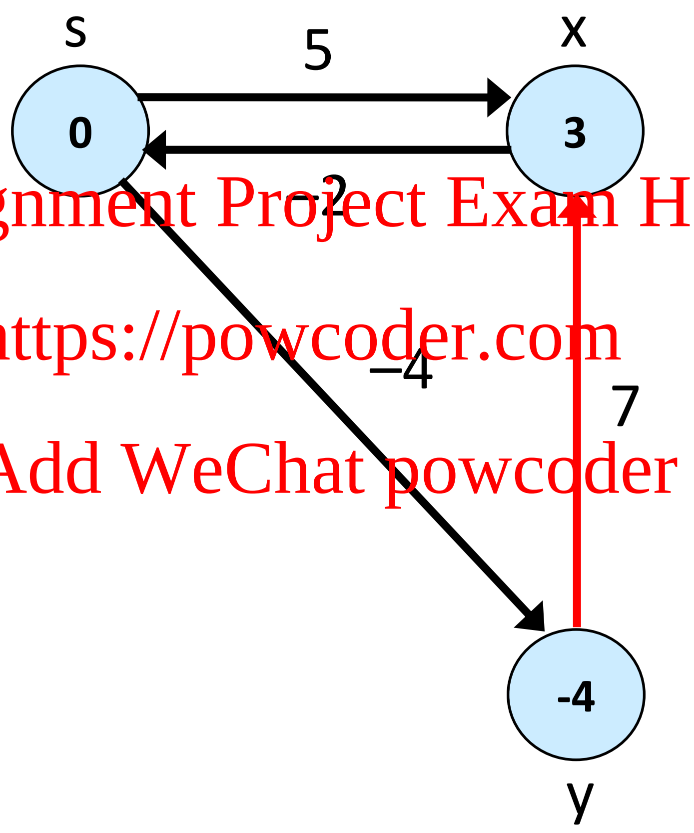
Example

Iteration 1



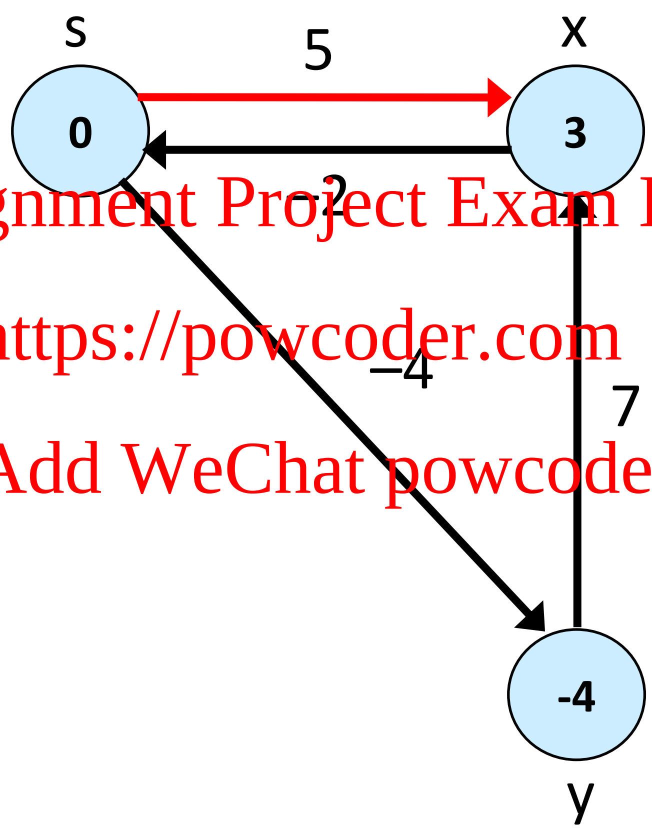
Example

Iteration 1



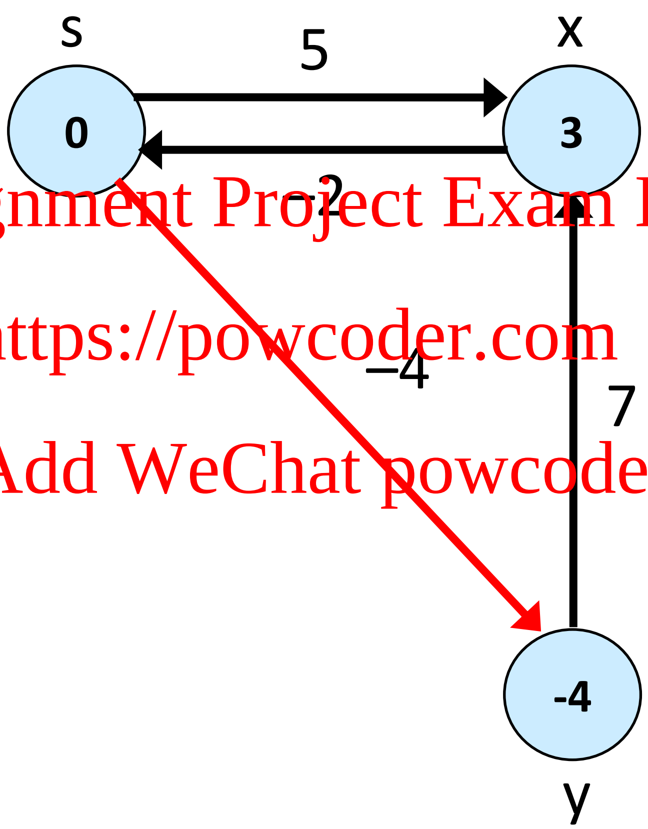
Example

Iteration 2



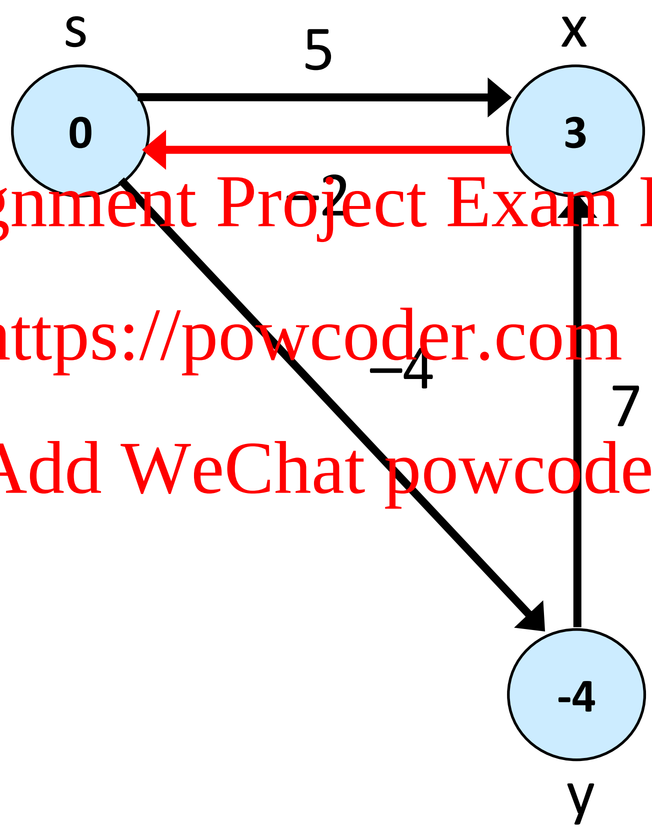
Example

Iteration 2



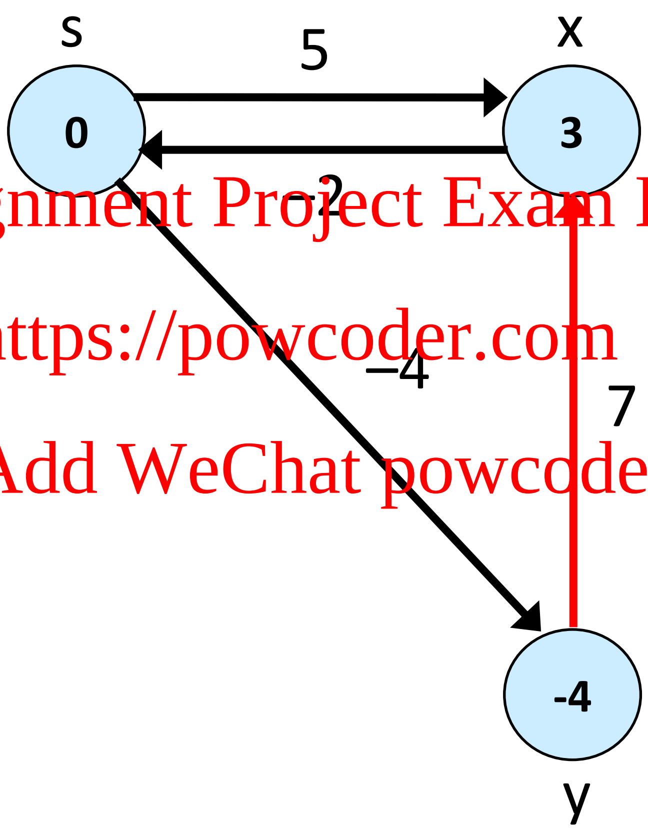
Example

Iteration 2

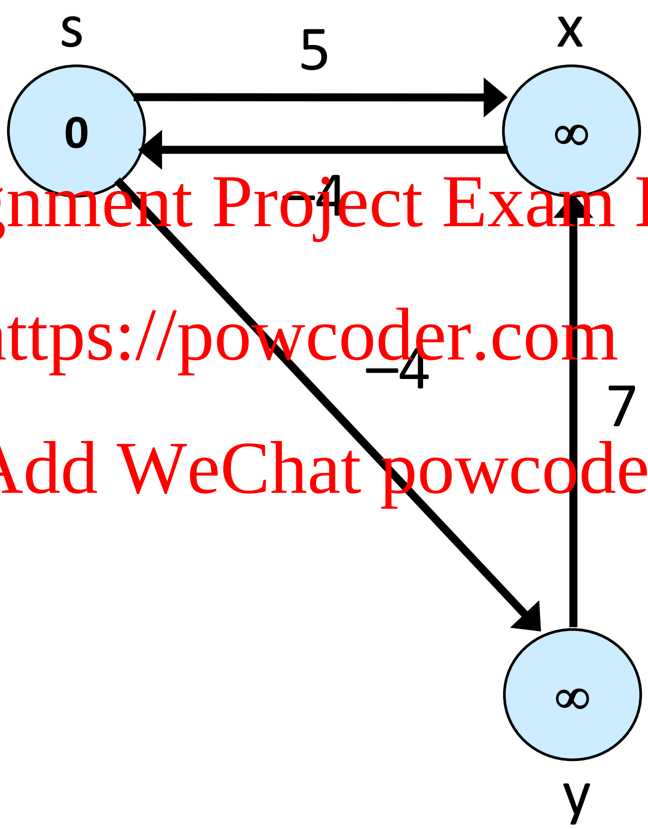


Example

Iteration 2

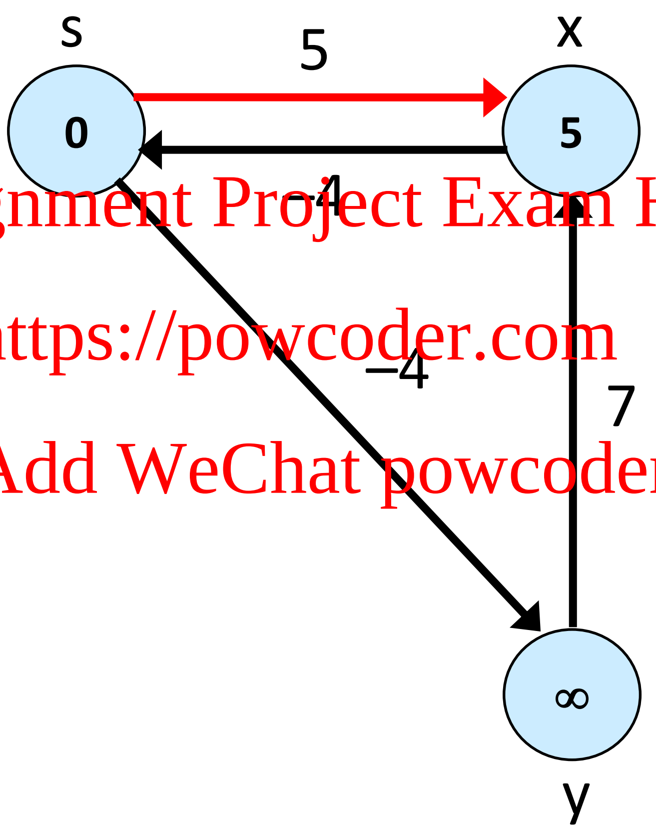


Example 2



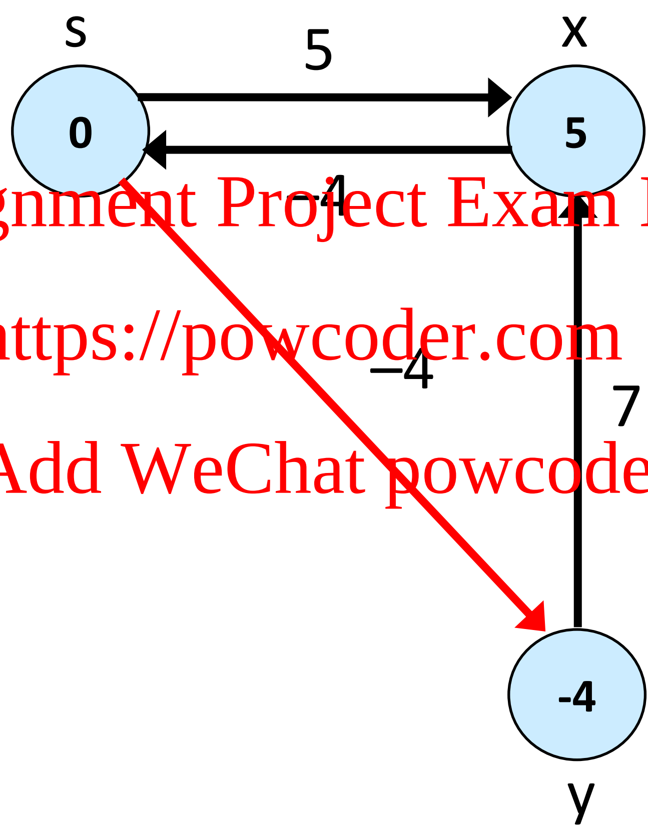
Example 2

Iteration 1



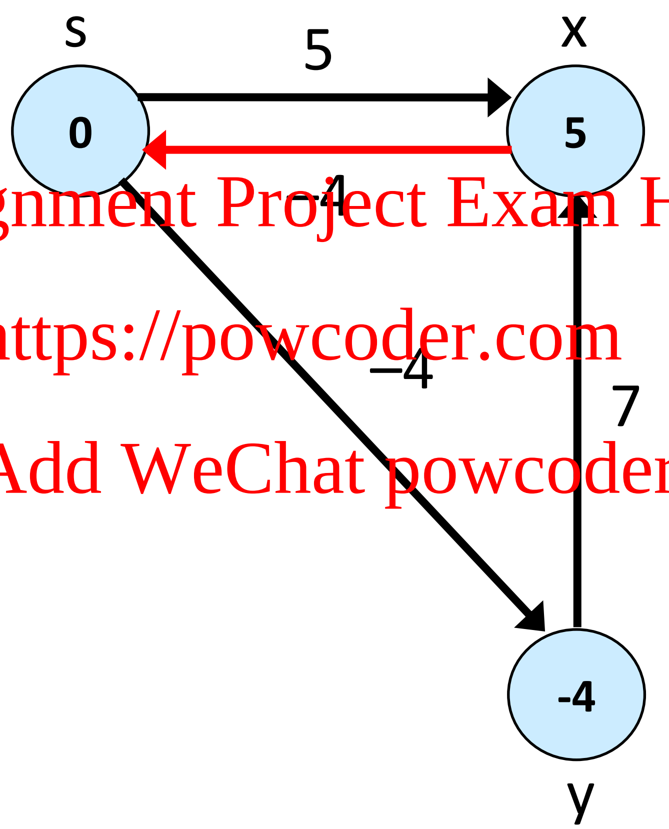
Example 2

Iteration 1



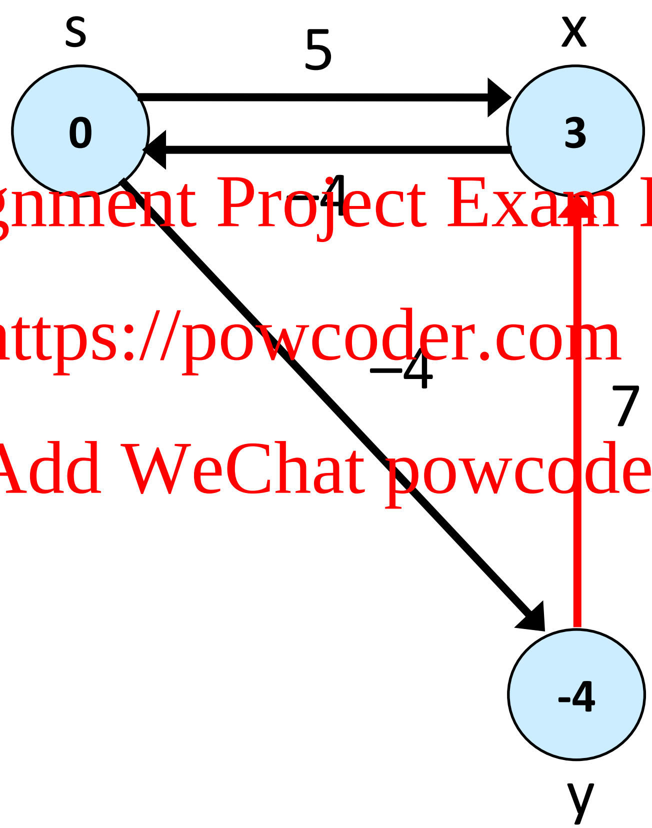
Example 2

Iteration 1



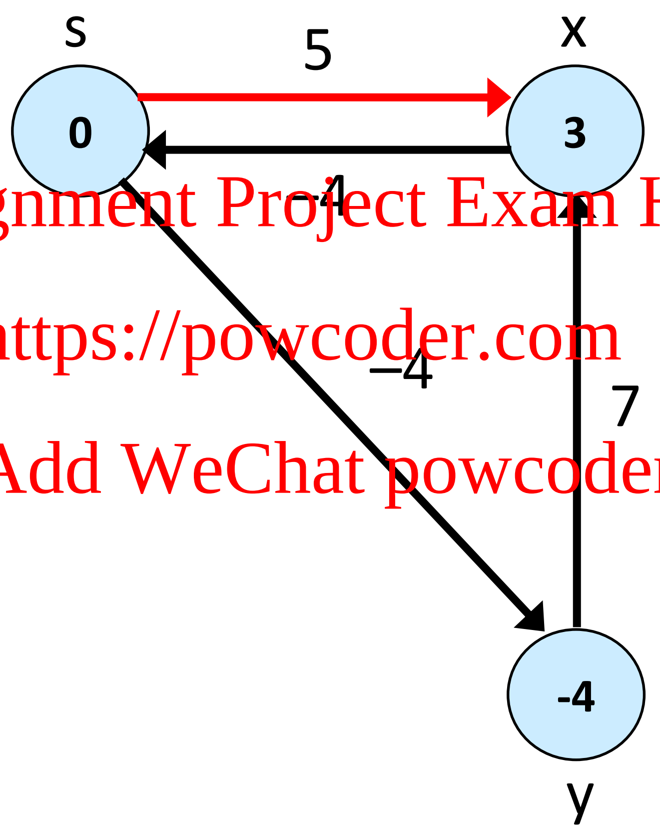
Example 2

Iteration 1



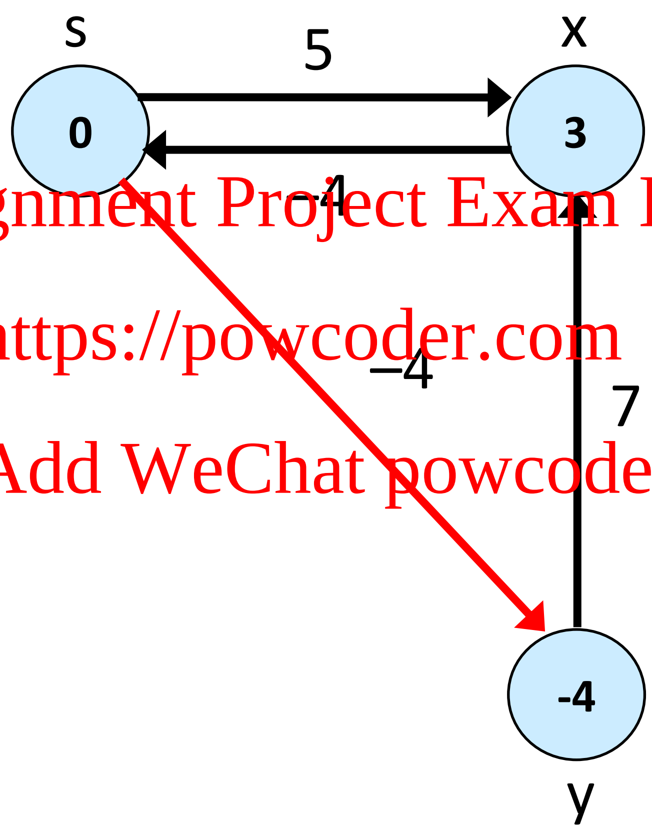
Example 2

Iteration 2



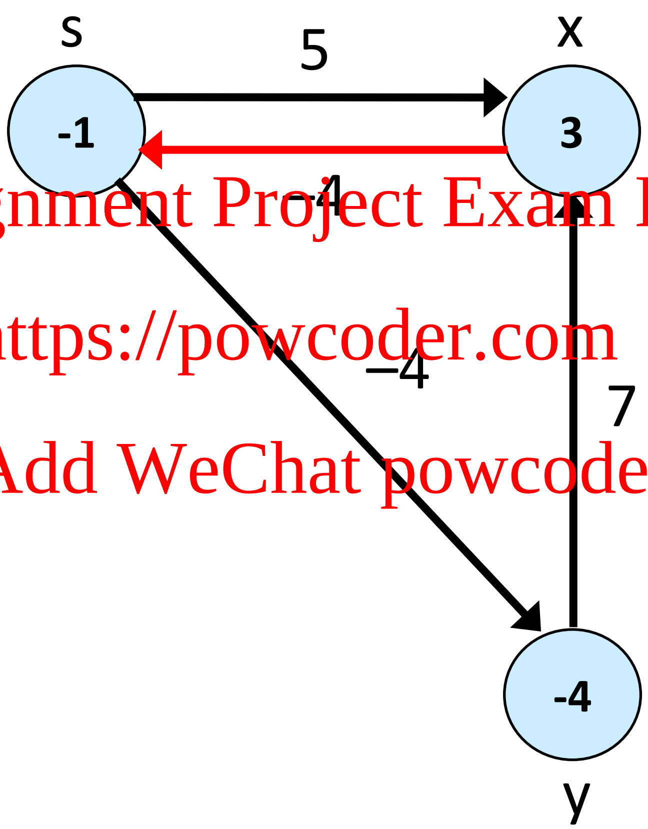
Example 2

Iteration 2



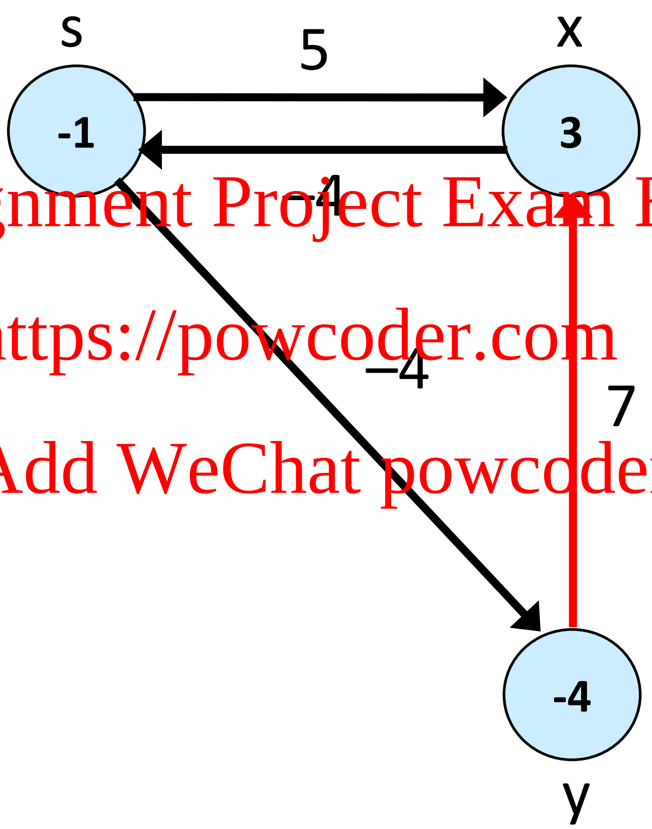
Example 2

Iteration 2



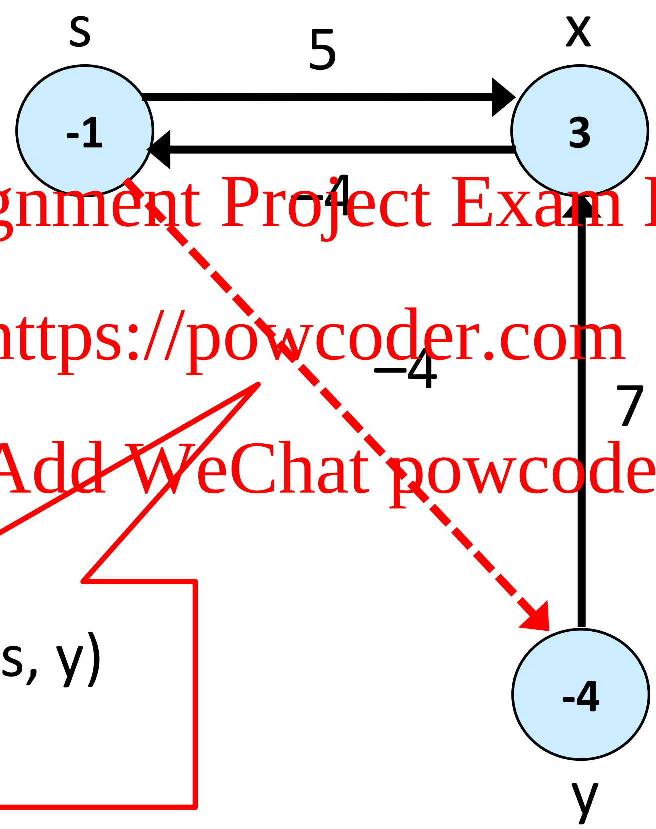
Example 2

Iteration 2



Example 2

Check



d[y] > d[s] + w(s, y)

$\Longrightarrow$ FALSE

Another Look at Bellman-Ford

Note: This is essentially **dynamic programming**.

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

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

i →

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

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.

False start…

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.

Conclusion. Need more subproblems!

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.

optimal substructure property
(proof via exchange argument)

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.

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

Dynamic programming algorithm

KNAPSACK (n, W, w_1, …, w_n, v_1, …, 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 $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w - w_i] \}.$

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

Example

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

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											

Example

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

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											

Example

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

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											

Example

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

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	6									
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

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

$M(i-1,w)$

Example

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

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											

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

$M(i-1,w)$

Example

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

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											

Example

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

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											

Example

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

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											

Example

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

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	[illegible]	[illegible]	[illegible]	[illegible]	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19	[illegible]	[illegible]	[illegible]	[illegible]	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.

weights are integers between 1 and W

Pf.

- 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]