

Introduction to Optimization

Dynamic Programming

November 18, 2016

École Centrale Paris, Châtenay-Malabry, France



Dimo Brockhoff
Inria Saclay – Ile-de-France

Course Overview

Date		Topic
Fri, 7.10.2016		Introduction
Fri, 28.10.2016	D	Introduction to Discrete Optimization + Greedy algorithms I
Fri, 4.11.2016	D	Greedy algorithms II + Branch and bound
Fri, 18.11.2016	D	Dynamic programming
Mon, 21.11.2016 in S103-S105	D	Approximation algorithms and heuristics
Fri, 25.11.2016 in S103-S105	C	Introduction to Continuous Optimization I
Mon, 28.11.2016	C	Introduction to Continuous Optimization II
Mon, 5.12.2016	C	Gradient-based Algorithms
Fri, 9.12.2016	C	Stochastic Optimization and Derivative Free Optimization I
Mon, 12.12.2016	C	Stochastic Optimization and Derivative Free Optimization II
Fri, 16.12.2016	C	Benchmarking Optimizers with the COCO platform
Wed, 4.1.2017		Exam

all classes last 3h15 and take place in S115-S117 (see exceptions)

Dynamic Programming

Wikipedia:

“[...] **dynamic programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems.”

But that's not all:

- dynamic programming also makes sure that the subproblems are not solved too often but only once by keeping the solutions of simpler subproblems in memory (“trading space vs. time”)
- it is an exact method, i.e. in comparison to the greedy approach, it always solves a problem to optimality

Two Properties Needed

Optimal Substructure

A solution can be constructed efficiently from optimal solutions of sub-problems

Overlapping Subproblems

Wikipedia: “[...] a problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.”

Note: in case of optimal substructure but independent subproblems, often greedy algorithms are a good choice; in this case, dynamic programming is often called “divide and conquer” instead

Main Idea Behind Dynamic Programming

Main idea: solve larger subproblems by breaking them down to smaller, easier subproblems in a recursive manner

Typical Algorithm Design:

- ① decompose the problem into subproblems and think about how to solve a larger problem with the solutions of its subproblems
- ② specify how you compute the value of a larger problem recursively with the help of the optimal values of its subproblems (“Bellman equation”)
- ③ bottom-up solving of the subproblems (i.e. computing their optimal value), starting from the smallest by using a table structure to store the optimal values and the Bellman equality (top-down approach also possible, but less common)
- ④ eventually construct the final solution (can be omitted if only the value of an optimal solution is sought)

Bellman Equation (aka “Principle of Optimality”)

- introduced by R. Bellman as “Principle of Optimality” in 1957
- the basic equation underlying dynamic programming
- necessary condition for optimality

citing Wikipedia:

“Richard Bellman showed that a dynamic optimization problem in **discrete time** can be stated in a **recursive**, step-by-step form known as **backward induction** by writing down the relationship between the value function in one period and the value function in the next period. The relationship between these two value functions is called the “Bellman equation”.”

- The value function here is the objective function.
- The Bellman equation exactly formalizes how to compute the optimal function value for a larger subproblem from the optimal function value of smaller subproblems.

we will see examples later today...

Why is it called “dynamic” and why “programming”?

- R. Bellman worked at the time, when he “invented” the idea, at the RAND Corporation who were strongly connected with the Air Force
- In order to avoid conflicts with the head of the Air Force at this time, R. Bellman decided against using terms like “mathematical” and he liked the word *dynamic* because it “has an absolutely precise meaning” and cannot be used “in a pejorative sense”
- in addition, it had the right meaning: “I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying.”
- Citing Wikipedia: “The word *programming* referred to the use of the method to find an optimal *program*, in the sense of a military schedule for training or logistics.”

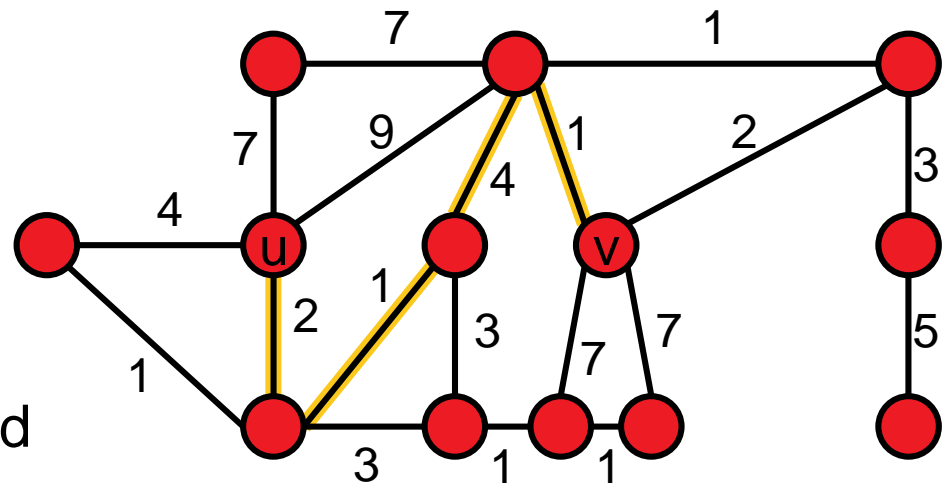
Reminder: Shortest Path Problem

Shortest Path problem:

Given a graph $G=(V,E)$ with edge weights w_i for each edge e_i . Find the shortest path from a vertex v to a vertex u , i.e., the path $(v, e_1=\{v, v_1\}, v_1, \dots, v_k, e_k=\{v_k, u\}, u)$ such that $w_1 + \dots + w_k$ is minimized.

Note:

We can often assume that the edge weights are stored in a distance matrix D of dimension $|V| \times |V|$ where an entry $D_{i,j}$ gives the weight between nodes i and j and “non-edges” are assigned a value of ∞



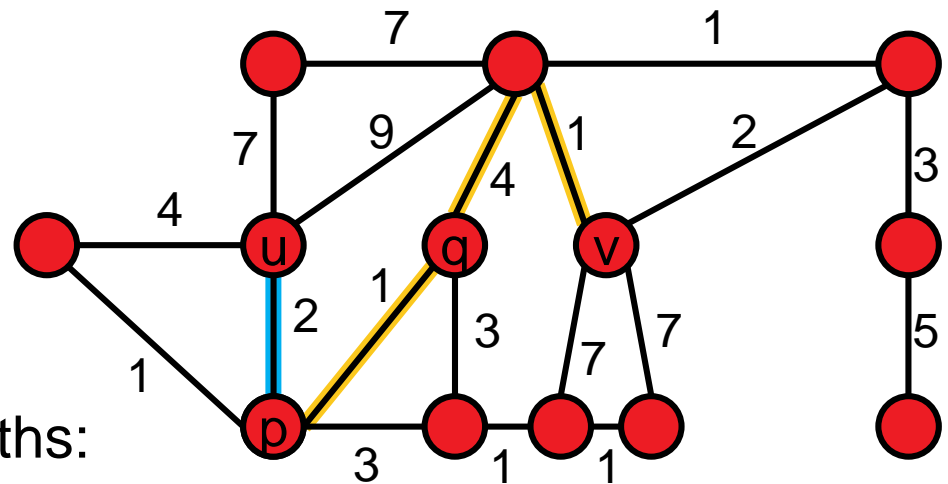
Opt. Substructure and Overlapping Subproblems

Optimal Substructure

The optimal path from u to v , if it contains another vertex p can be constructed by simply joining the optimal path from u to p with the optimal path from p to v .

Overlapping Subproblems

Optimal shortest sub-paths can be reused when computing longer paths:
e.g. the optimal path from u to p is contained in the optimal path from u to q and in the optimal path from u to v .



The Algorithm of E. Dijkstra (1956)

ShortestPathDijkstra(G, D, source, target):

Initialization:

- $\text{dist}(\text{source}) = 0$ and for all $v \in V$: $\text{dist}(v) = \infty$
- for all $v \in V$: $\text{prev}(v) = \text{null}$ # predecessors on opt. path
- $U = V$ # U: unexplored vertices

Unless U empty do:

- $\text{newNode} = \text{argmin}_{u \in U} \{\text{dist}(u)\}$
- remove newNode from U
- for each neighbor v of newNode do:
 - $\text{altDist} = \text{dist}(\text{newNode}) + D_{\text{newNode},v}$
 - if $\text{altDist} < \text{dist}(v)$:
 - $\text{dist}(v) = \text{altDist}$
 - $\text{prev}(v) = u$

Very Short Exercise

Question:

Is Dijkstra's algorithm a dynamic programming algorithm?

Answer:

- that is a tricky question ;-)
- it has greedy elements, but also stores the answers to subproblems without recomputing them
- so, actually, it is a dynamic programming algorithm with a greedy selection of the next subproblem to be computed

The Algorithm of R. Floyd (1962)

Idea:

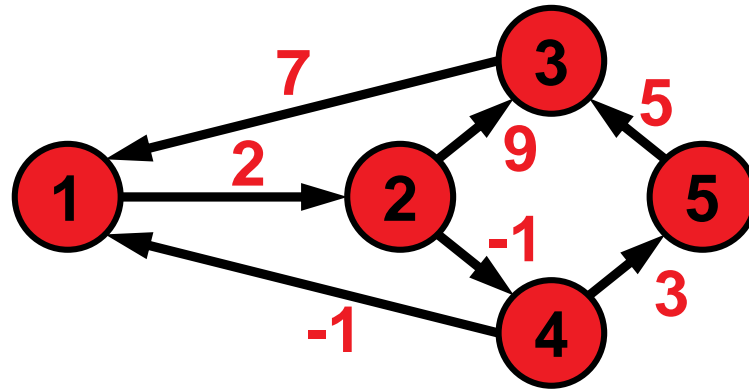
- if we knew that the shortest path between source and target goes through node v , we would be able to construct the optimal path from the shorter paths “source $\rightarrow v$ ” and “ $v \rightarrow$ target”
- subproblem $P(k)$: compute all shortest paths where the intermediate nodes can be chosen from v_1, \dots, v_k

ShortestPathFloyd($G, D, \text{source}, \text{target}$) [= AllPairsShortestPath(G)]

- Init: for all $1 \leq i, j \leq |V|$: $\text{dist}(i, j) = D_{i, j}$
- For $k = 1$ to $|V|$ # solve subproblems $P(k)$
 - for all pairs of nodes (i.e. $1 \leq i, j \leq |V|$):
 - $\text{dist}(i, j) = \min \{ \text{dist}(i, j), \text{dist}(i, k) + \text{dist}(k, j) \}$

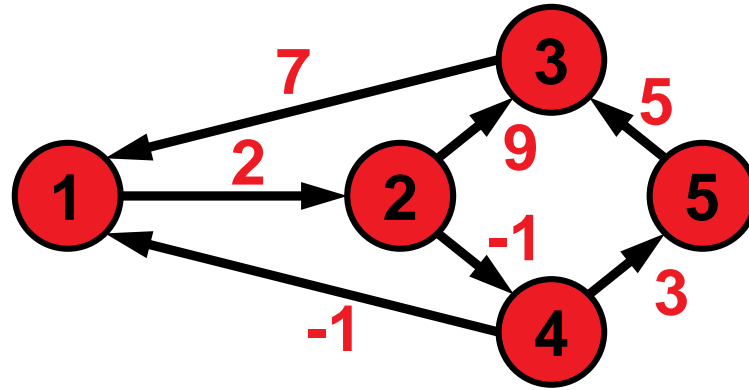
Note: This algorithm has the advantage that it can handle negative weights as long as no cycle with negative total weight exists

Example



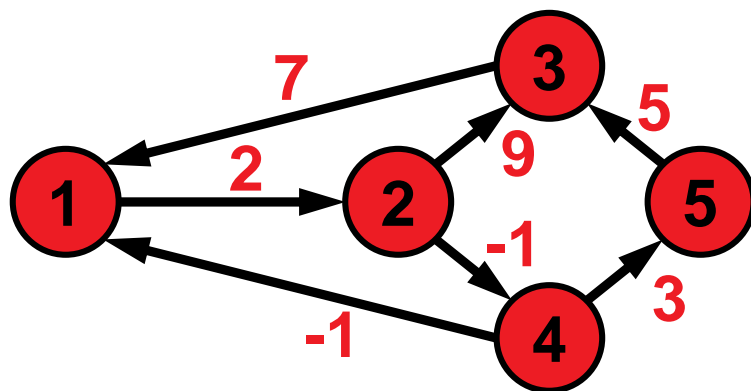
k=0	1	2	3	4	5
1					
2					
3					
4					
5					

Example



k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

Example



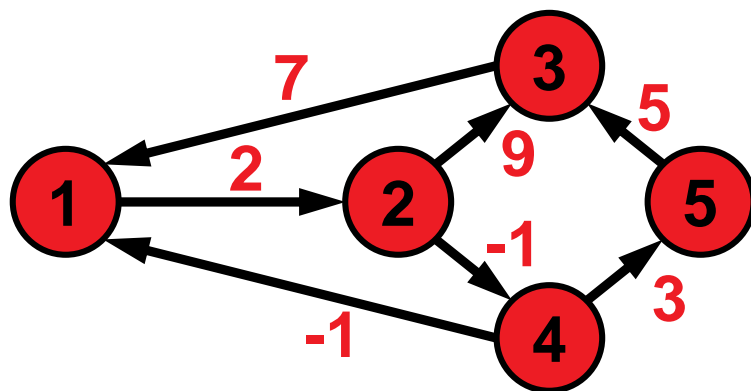
allow 1 as intermediate node



k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

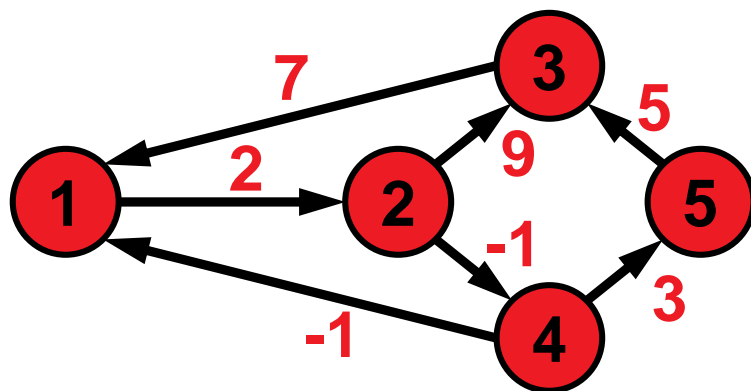


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

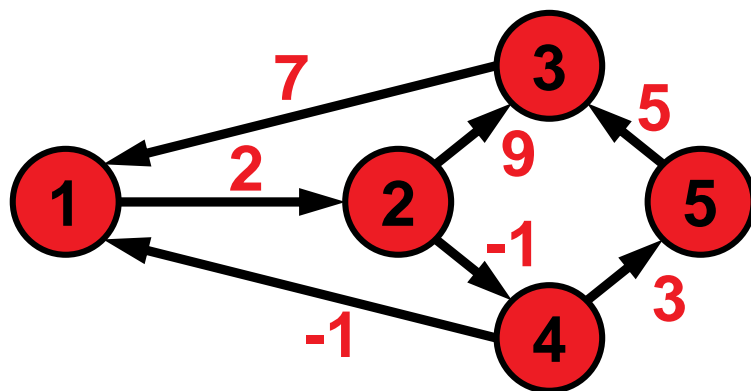


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

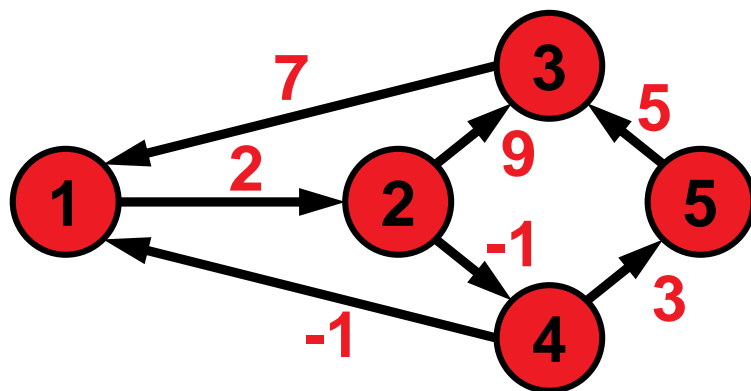


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3		9			
4		1			
5					

Example

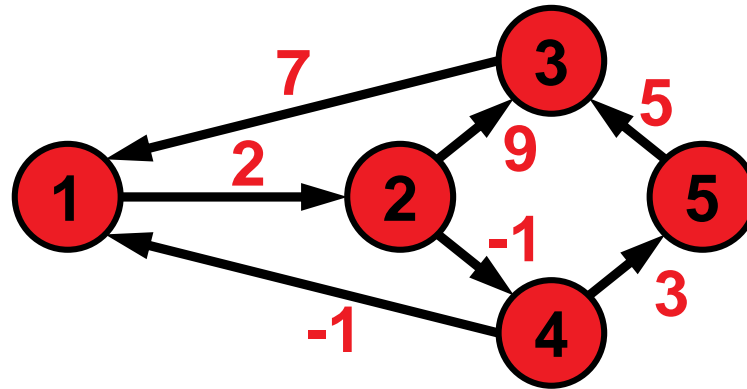


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

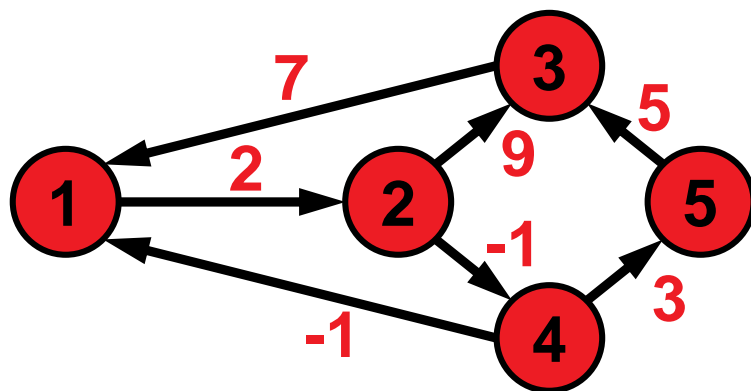
Example



allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5	k=2	1	2	3	4	5
1	∞	2	∞	∞	∞	1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞	2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞	3	7	9	∞	∞	∞
4	-1	1	∞	∞	3	4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞	5	∞	∞	5	∞	∞

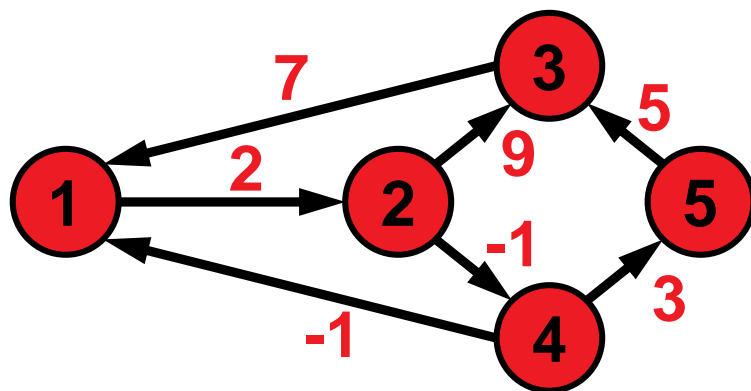
Example



allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5	k=2	1	2	3	4	5
1	∞	2	∞	∞	∞	1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞	2	∞	∞	9	-1	∞
3	∞	9	∞	∞	∞	3	7	9	∞	∞	∞
4	-1	1	∞	∞	3	4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞	5	∞	∞	5	∞	∞

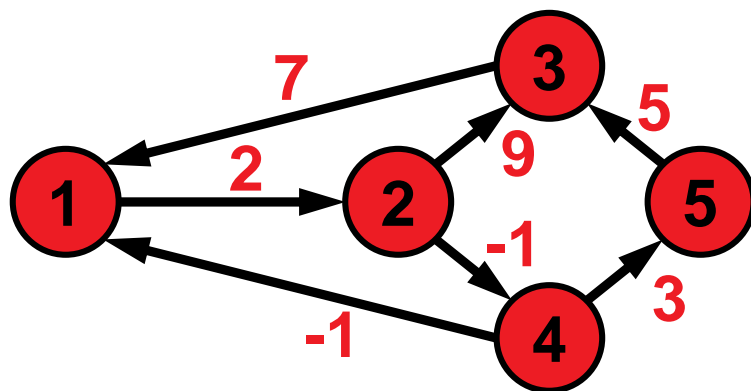
Example



allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5	k=2	1	2	3	4	5
1	∞	2	∞	∞	∞	1	∞	2	11	1	∞
2	∞	∞	9	-1	∞	2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞	3	7	9	18	8	∞
4	-1	1	∞	∞	3	4	-1	1	10	0	3
5	∞	∞	5	∞	∞	5	∞	∞	5	∞	∞

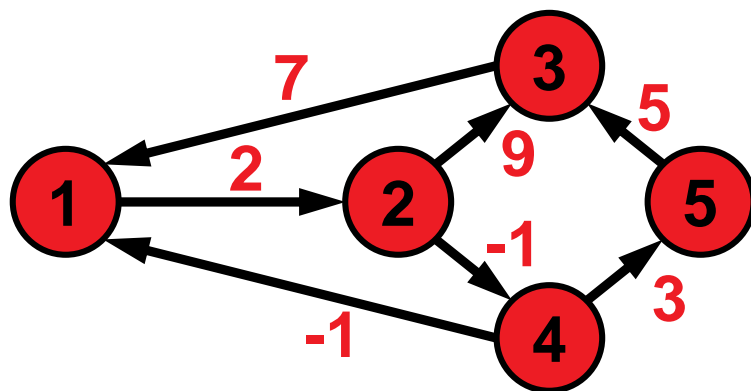
Example



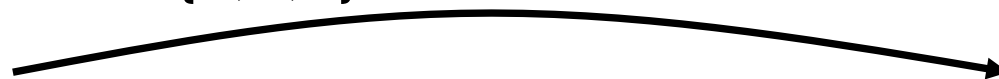
allow $\{1,2,3\}$ as intermediate nodes

$k=2$	1	2	3	4	5	$k=3$	1	2	3	4	5
1	∞	2	11	1	∞	1	∞	2	11	1	∞
2	∞	∞	9	-1	∞	2	∞	∞	9	-1	∞
3	7	9	18	8	∞	3	7	9	18	8	∞
4	-1	1	10	0	3	4	-1	1	10	0	3
5	∞	∞	5	∞	∞	5	∞	∞	5	∞	∞

Example

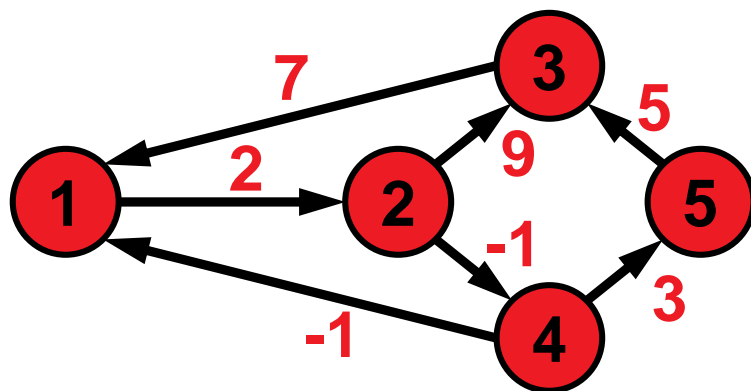


allow $\{1,2,3\}$ as intermediate nodes



k=2	1	2	3	4	5	k=3	1	2	3	4	5
1	∞	2	11	1	∞	1			11		∞
2	∞	∞	9	-1	∞	2			9		∞
3	7	9	18	8	∞	3	7	9	18	8	∞
4	-1	1	10	0	3	4			10		3
5	∞	∞	5	∞	∞	5			5		∞

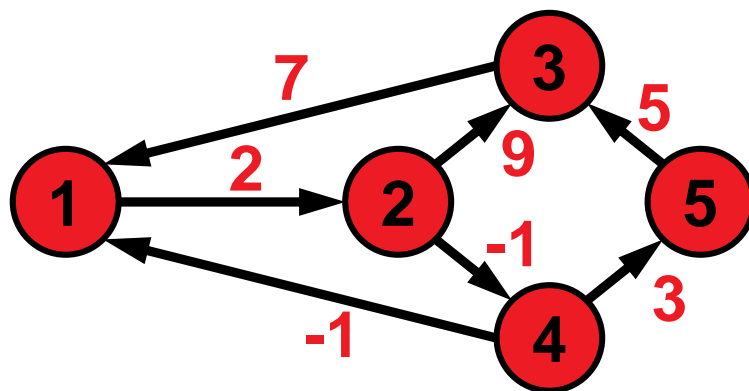
Example



allow $\{1,2,3\}$ as intermediate nodes

$k=2$	1	2	3	4	5	$k=3$	1	2	3	4	5
1	∞	2	11	1	∞	1	18	2	11	1	∞
2	∞	∞	9	-1	∞	2	16	18	9	-1	∞
3	7	9	18	8	∞	3	7	9	18	8	∞
4	-1	1	10	0	3	4	-1	1	10	0	3
5	∞	∞	5	∞	∞	5	12	14	5	13	∞

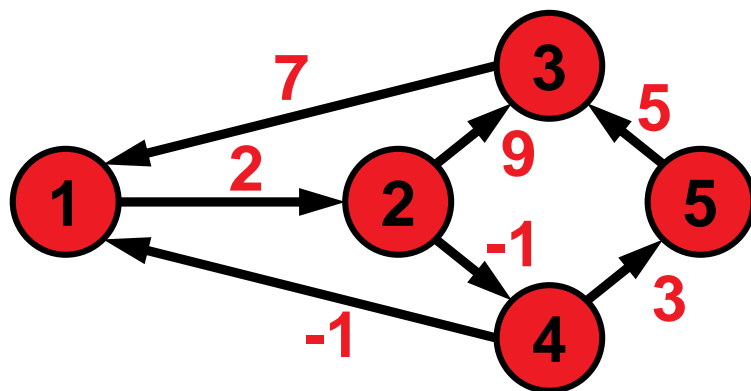
Example



allow $\{1,2,3,4\}$ as intermediate nodes

k=3	1	2	3	4	5	k=4	1	2	3	4	5
1	18	2	11	1	∞	1	18	2	11	1	∞
2	16	18	9	-1	∞	2	16	18	9	-1	∞
3	7	9	18	8	∞	3	7	9	18	8	∞
4	-1	1	10	0	3	4	-1	1	10	0	3
5	12	14	5	13	∞	5	12	14	5	13	∞

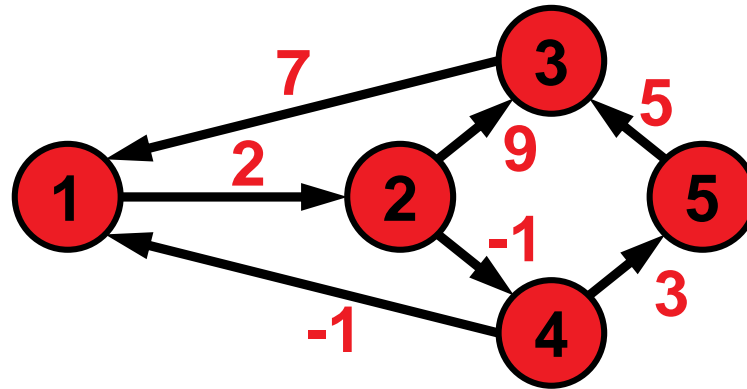
Example



allow $\{1,2,3,4\}$ as intermediate nodes

k=3	1	2	3	4	5	k=4	1	2	3	4	5
1	18	2	11	1	∞	1				1	
2	16	18	9	-1	∞	2				-1	
3	7	9	18	8	∞	3				8	
4	-1	1	10	0	3	4	-1	1	10	0	3
5	12	14	5	13	∞	5				13	

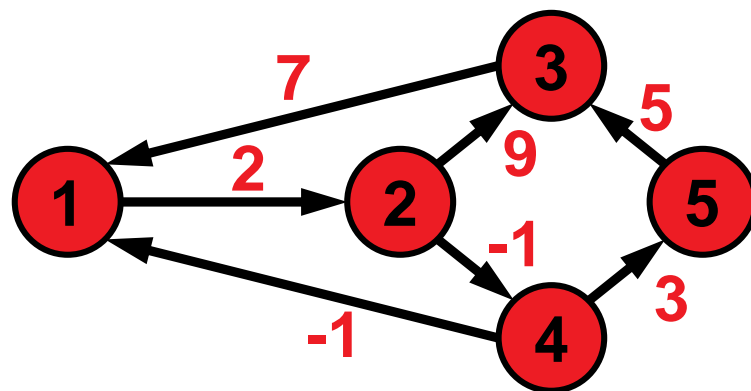
Example



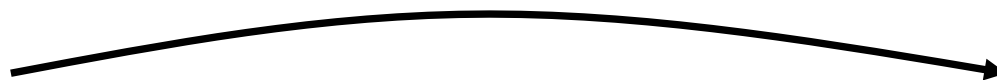
allow $\{1,2,3,4\}$ as intermediate nodes

k=3	1	2	3	4	5	k=4	1	2	3	4	5
1	18	2	11	1	∞	1	0	2	11	1	4
2	16	18	9	-1	∞	2	-2	0	9	-1	2
3	7	9	18	8	∞	3	7	9	18	8	11
4	-1	1	10	0	3	4	-1	1	10	0	3
5	12	14	5	13	∞	5	12	14	5	13	16

Example



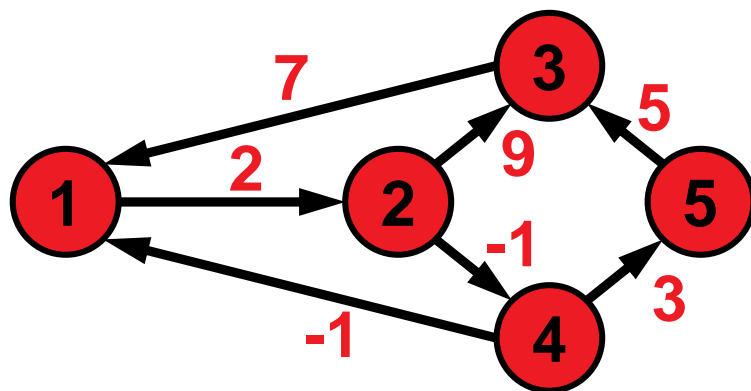
allow all nodes as intermediate nodes



k=4	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

k=5	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

Example



allow all nodes as intermediate nodes

k=4	1	2	3	4	5	k=5	1	2	3	4	5
1	0	2	11	1	4	1	0	2	9	1	4
2	-2	0	9	-1	2	2	-2	0	7	-1	2
3	7	9	18	8	11	3	7	9	16	8	11
4	-1	1	10	0	3	4	-1	1	8	0	3
5	12	14	5	13	16	5	12	14	5	13	16

Runtime Considerations and Correctness

$O(|V|^3)$ easy to show

- $O(|V|^2)$ many distances need to be updated $O(|V|)$ times

Correctness

- given by the Bellman equation
$$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$$
- only correct if cycles do not have negative total weight (can be checked in final distance matrix if diagonal elements are negative)

But How Can We Actually Construct the Paths?

- Construct matrix of predecessors P alongside distance matrix
- $P_{i,j}(k)$ = predecessor of node j on path from i to j (at algo. step k)
- no extra costs (asymptotically)

$$P_{i,j}(0) = \begin{cases} 0 & \text{if } i = j \text{ or } d_{i,j} = \infty \\ i & \text{in all other cases} \end{cases}$$

$$P_{i,j}(k) = \begin{cases} P_{i,j}(k-1) & \text{if } \text{dist}(i,j) \leq \text{dist}(i,k) + \text{dist}(k,j) \\ P_{k,j}(k-1) & \text{if } \text{dist}(i,j) > \text{dist}(i,k) + \text{dist}(k,j) \end{cases}$$

Exercise:

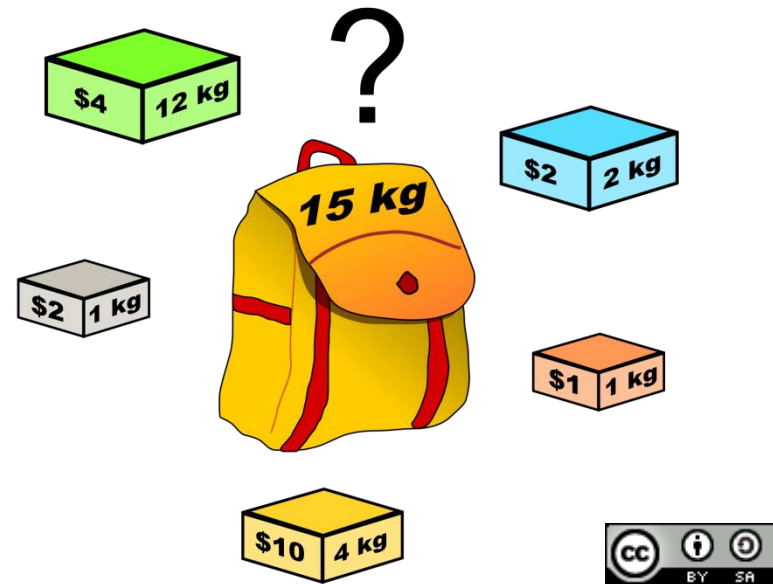
The Knapsack Problem and Dynamic Programming

Exercise: Dynamic Programming for 0-1 KP

0-1 Knapsack Problem (KP)

$$\max. \sum_{j=1}^n p_j x_j \text{ with } x_j \in \{0, 1\}$$

$$\text{s.t. } \sum_{j=1}^n w_j x_j \leq W$$



Goal: a dynamic programming algorithm for KP

Questions:

- what could be subproblems?
- how to solve subproblems with the help of smaller ones?
- how to solve the smallest subproblems exactly?

Exercise:

The Knapsack Problem and Dynamic Programming

```
http://chercheurs.lille.inria.fr/  
~brockhof/introoptimization/exercises.php
```

Opt. Substructure and Overlapping Subproblems

Consider the following subproblem:

$P(i, j)$: optimal profit when allowed to pack *only the first i items* into a knapsack of size j

Questions:

- a) what could be subproblems?
- b) how to solve subproblems with the help of smaller ones?
[write the Bellman equation!]
- c) how to solve the smallest subproblems exactly?

Opt. Substructure and Overlapping Subproblems

Consider the following subproblem:

$P(i, j)$: optimal profit when allowed to pack *only the first i items* into a knapsack of size j

Optimal Substructure

The optimal choice of whether taking item i or not can be made easily for a knapsack of weight j if we know the optimal choice for items $1 \dots i - 1$:

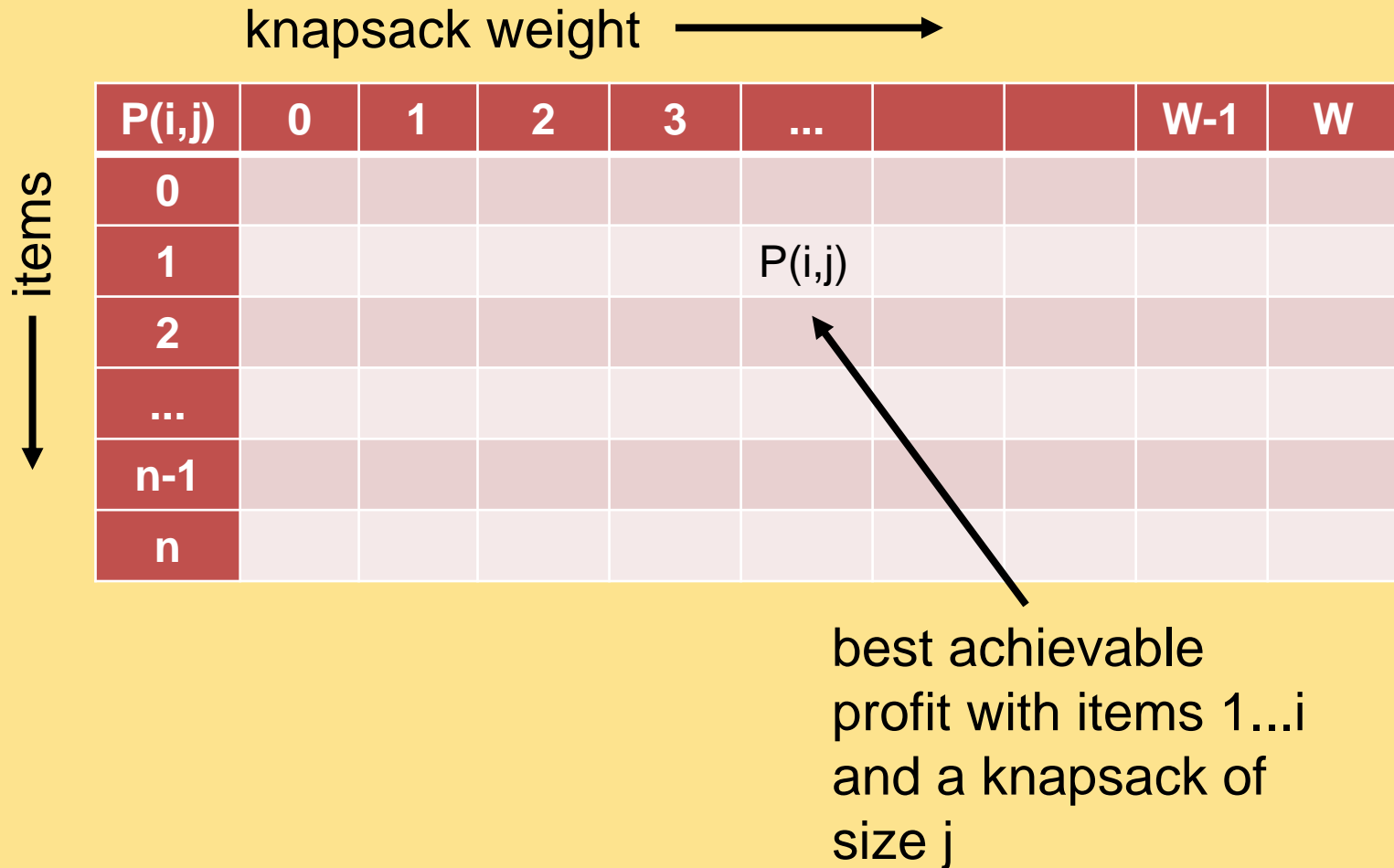
$$P(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Overlapping Subproblems

a recursive implementation of the Bellman equation is simple, but the $P(i, j)$ might need to be computed more than once!

Dynamic Programming Approach to the KP

To circumvent computing the subproblems more than once, we can store their results (in a matrix for example)...



Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W=11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0												
1												
2												
3												
4												
5												

initialization:

$$P(i,j) = 0 \text{ if } i = 0 \text{ or } j = 0$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W=11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											

initialization:

$$P(i,j) = 0 \text{ if } i = 0 \text{ or } j = 0$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0										
2	0											
3	0											
4	0											
5	0											

items \downarrow

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0										
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

items \downarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0									
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0							
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

items \downarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4						
2	0											
3	0											
4	0											
5	0											

A red arrow points from the cell (1,5) to the cell (0,5). A blue arrow points from the cell (1,5) to the cell (1,4). A red annotation $+p_1 (= 4)$ is placed below the cell (1,4).

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

items \downarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4					
2	0											
3	0											
4	0											
5	0											

A red arrow points from the cell (1,6) to (1,5) with the label $+p_1(=4)$. A blue arrow points from the cell (1,5) to (1,6).

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

	P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
items	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	4	4	4	4	4	4	4
2	0												
3	0												
4	0												
5	0												

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

items \downarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4					
3	0											
4	0											
5	0											

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits
 (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

items \downarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10				
3	0											
4	0											
5	0											

A red arrow points from the cell (2,7) to (1,1). A blue arrow points from the cell (2,7) to (2,8). The text $+p_2 (= 10)$ is written in red below the cell (2,7).

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0											
4	0											
5	0											

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3							
4	0											
5	0											

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4						
4	0											
5	0											

items \downarrow

Annotations: A red arrow points from the cell (3,3) to (2,5) with the text $+p_3 (= 3)$. A blue arrow points from the cell (2,5) to (3,5).

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

items \downarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4					
4	0											
5	0											

Annotations: A red arrow points from the value 4 at (i=3, j=5) to the value 4 at (i=2, j=6) with the label $+p_3 (= 3)$. A blue arrow points from the value 4 at (i=2, j=6) to the value 4 at (i=3, j=7). The cell at (i=3, j=7) is highlighted in green.

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits
 (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	etc.			
4	0											
5	0											

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits
 (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is $W = 11$.

knapsack weight \longrightarrow

$P(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Question: How to obtain the actual packing?

Answer: we just need to remember where the max came from!

knapsack weight \longrightarrow

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

items \downarrow

Annotations: $x_1 = 0$ (blue arrow from (1,0) to (1,1)), $x_2 = 1$ (red arrow from (2,7) to (1,7)), $x_3 = 0$ (blue arrow from (3,7) to (3,8)), $x_4 = 1$ (red arrow from (4,10) to (3,10)), $x_5 = 0$ (blue arrow from (5,11) to (5,10)).

for $i = 1$ to n :

for $j = 1$ to W :

$$P(i, j) = \begin{cases} P(i - 1, j) & \text{if } w_i > j \\ \max\{P(i - 1, j), p_i + P(i - 1, j - w_i)\} & \text{if } w_i \leq j \end{cases}$$

Conclusions

I hope it became clear...

- ...what the algorithm design idea of **dynamic programming** is
- ...for which problem types it is supposed to be **suitable**
- ...and how to **apply** the idea to the **knapsack problem**