Introduction to Optimization

Dynamic Programming

November 2, 2015

École Centrale Paris, Châtenay-Malabry, France

Dimo Brockhoff

INRIA Lille – Nord Europe

Course Overview

Date		Topic
Mon, 21.9.2015		Introduction
Mon, 28.9.2015	D	Basic Flavors of Complexity Theory
Mon, 5.10.2015	D	Greedy algorithms
Mon, 12.10.2015	D	Branch and bound (switched w/ dynamic programming)
Mon, 2.11.2015	**D**	**Dynamic programming** *[salle Proto]*
Fri, 6.11.2015	D	Approximation algorithms and heuristics *[S205/S207]*
Mon, 9.11.2015	C	Introduction to Continuous Optimization I *[S118]*
Fri, 13.11.2015	C	Introduction to Continuous Optimization II *[from here onwards always: S205/S207]*
Fri, 20.11.2015	C	Gradient-based Algorithms
Fri, 27.11.2015	C	End of Gradient-based Algorithms + Linear Programming
Fri, 4.12.2015	C	Stochastic Optimization and Derivative Free Optimization
Tue, 15.12.2015		Exam

all classes + exam last 3 hours (incl. a 15min break)

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:

❶ decompose the problem into subproblems and think about how to solve a larger problem with the solutions of its subproblems

❷ specify how you compute the value of a larger problem recursively with the help of the optimal values of its subproblems ("Bellman equation")

❸ 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)

❹ 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...

Historical Note

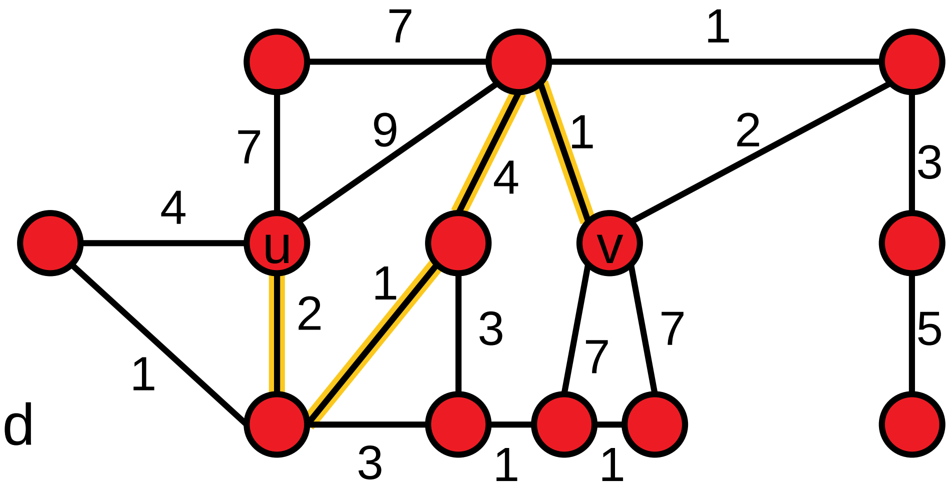
Why is it called "dynamic" and why "programming"?

- 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, ..., v_k, $e_k=\{v_k,u\}$, u) such that $w_1 + ... + w_k$ is minimized.



Note:

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

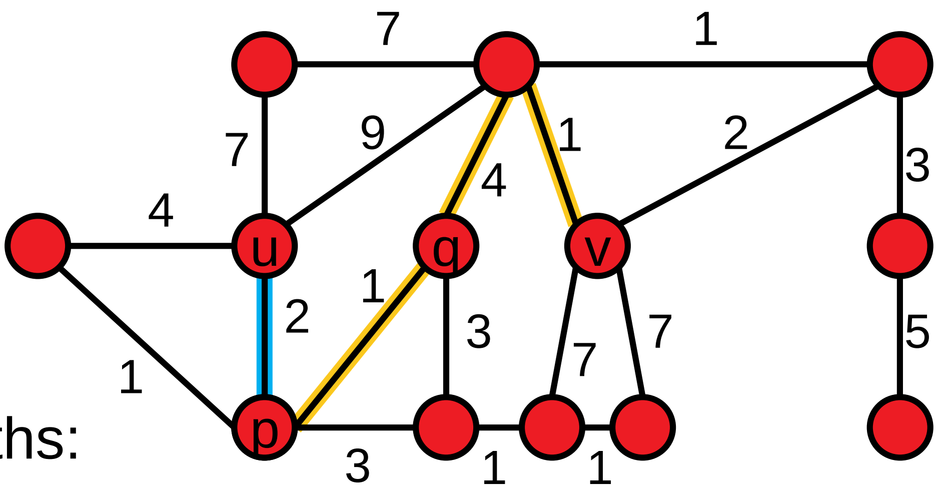
Opt. Substructure and Overlapping Subproblems

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:
    ▪ dist(source) = 0 and for all v $\in$ V: dist(v)= $\infty$
    ▪ for all v $\in$ V: prev(v) = null          # predecessors on opt. path
    ▪ U = V                                     # U: unexplored vertices
Unless U empty do:
    ▪ newNode = argmin$_{u \in U}$ {dist(u)}
    ▪ remove newNode from U
    ▪ for each neighbor v of newNode do:
        ▪ altDist = dist(newNode) + $D_{newNode,v}$
        ▪ if altDist < dist(v):
            ▪ dist(v) = altDist
            ▪ 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:

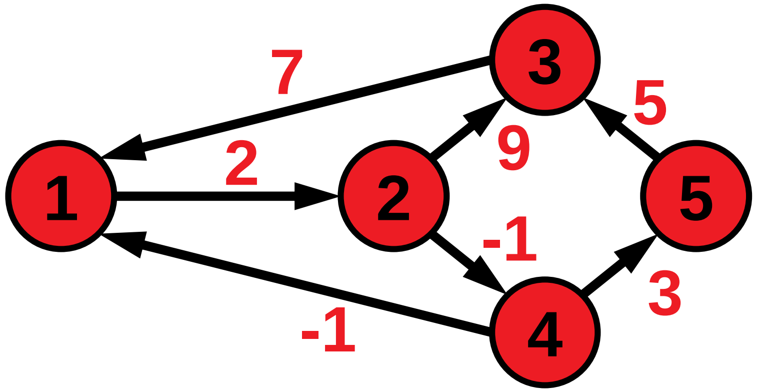
- if we knew that the shortest path between source and target goes through node v, we would be able to construct the optimal path from the shorter paths “source→v” and “v→target”
- subproblem P(k): compute all shortest paths where the intermediate nodes can be chosen from $v_1, ..., v_k$

ShortestPathFloyd(G, D, source, target) [= AllPairsShortestPath(G)]

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

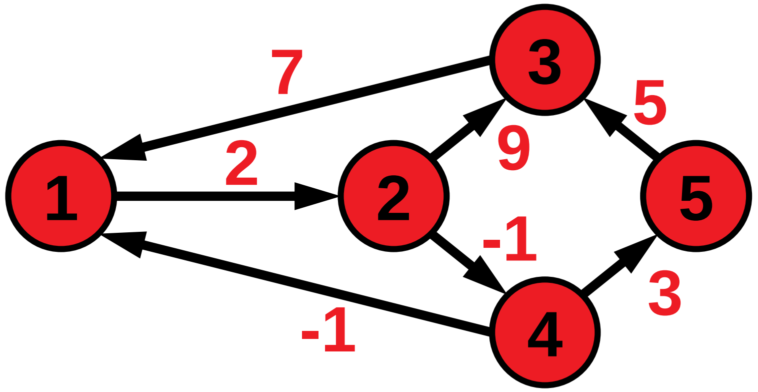
Note: This algorithm has the advantage that it can handle negative weights as long as no cycle with negative total weight exists

Example



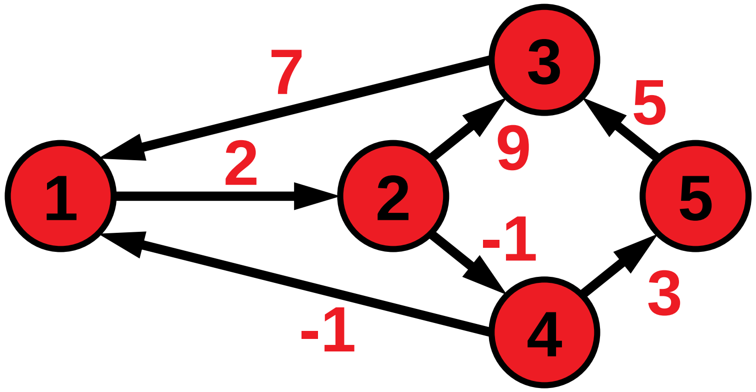
k=0	1	2	3	4	5
1					
2					
3					
4					
5					

Example



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

Example

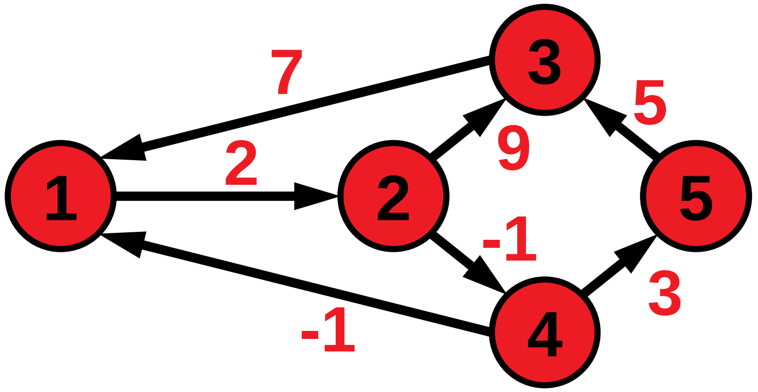


allow 1 as intermediate node

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

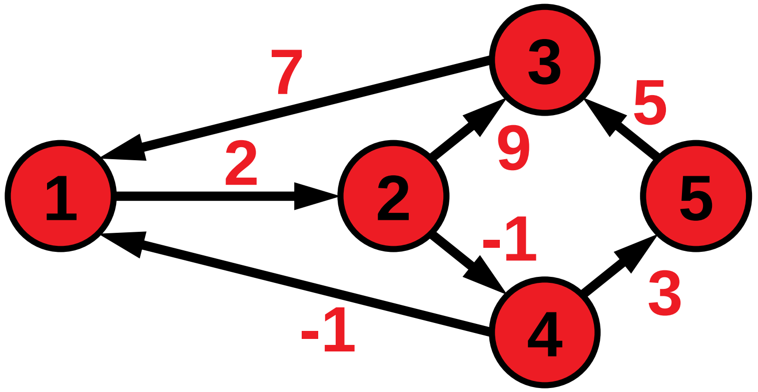


allow 1 as intermediate node

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

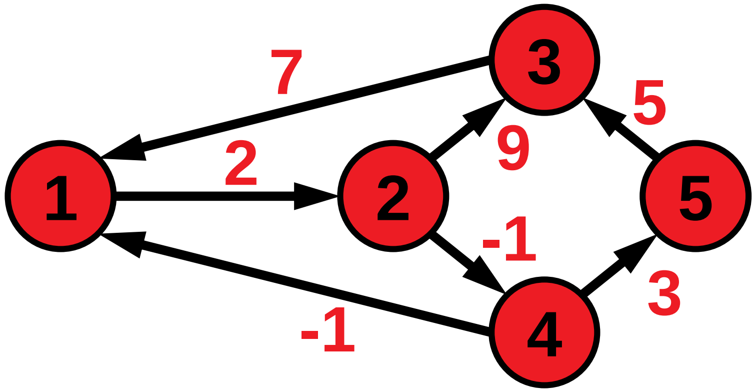


allow 1 as intermediate node

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

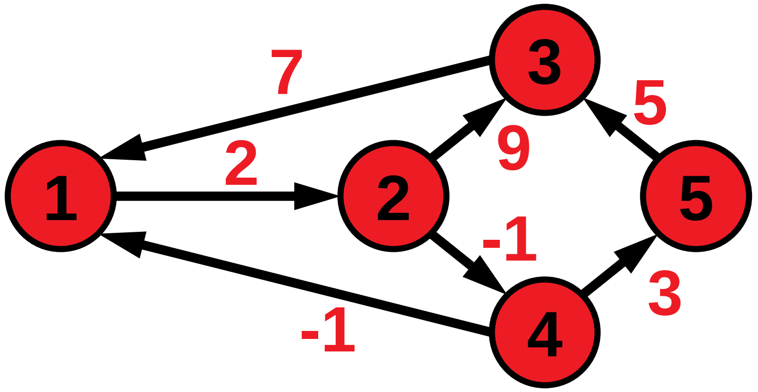


allow 1 as intermediate node

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

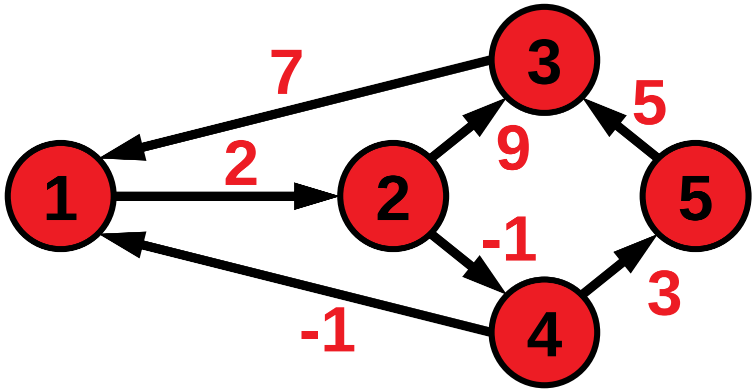


allow 1 as intermediate node

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

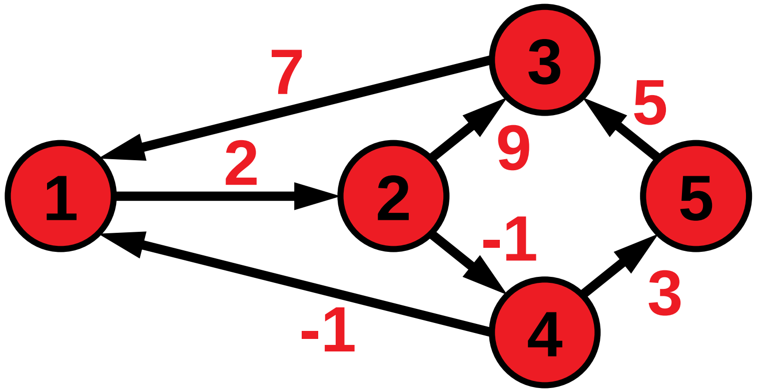


allow 1 & 2 as intermediate nodes

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

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

Example

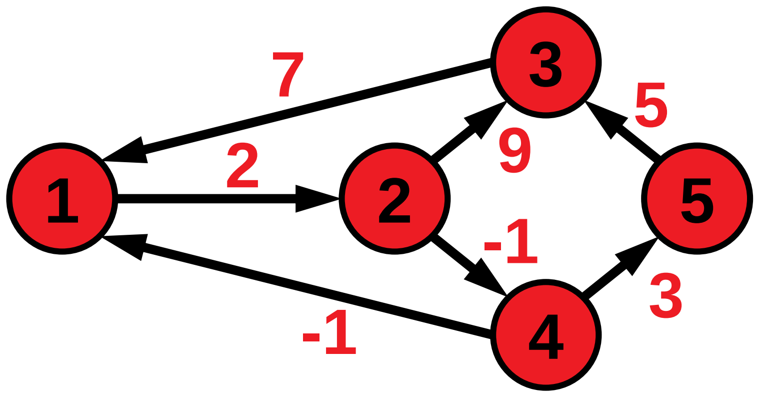


allow 1 & 2 as intermediate nodes

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

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

Example

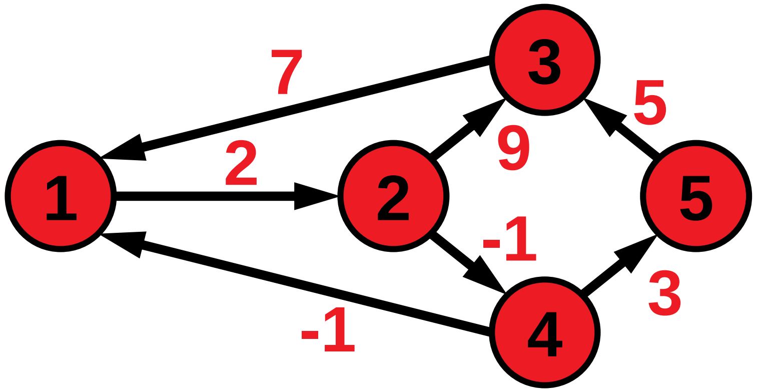


allow 1 & 2 as intermediate nodes

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

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

Example

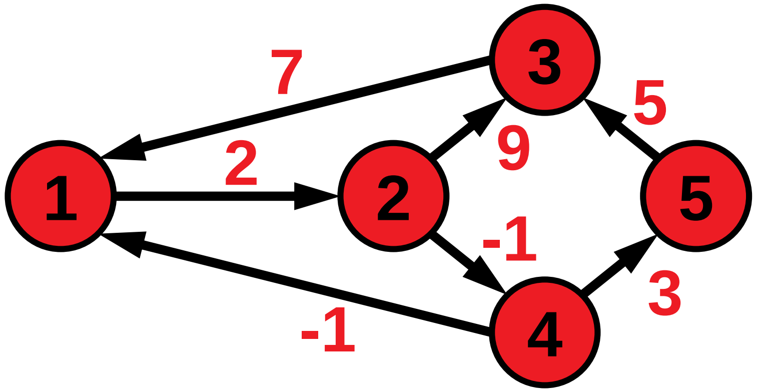


allow {1,2,3} as intermediate nodes

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

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

Example

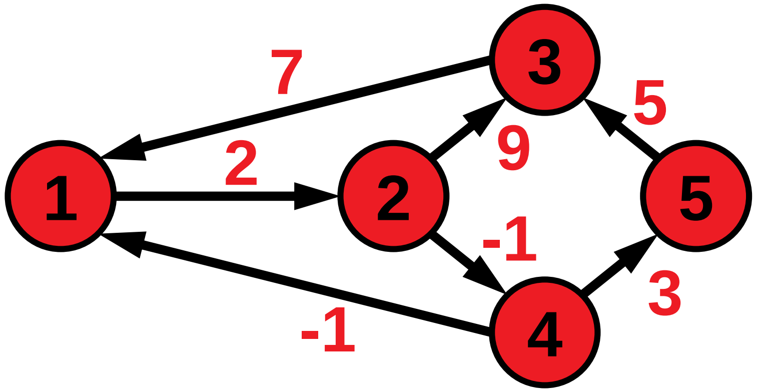


allow {1,2,3} as intermediate nodes

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

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

Example

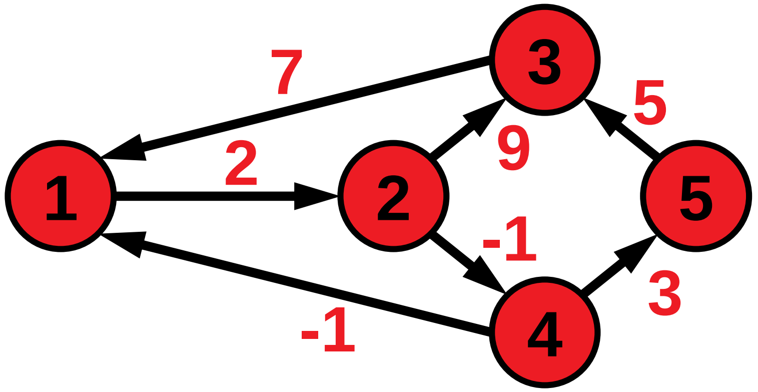


allow {1,2,3} as intermediate nodes

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

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

Example



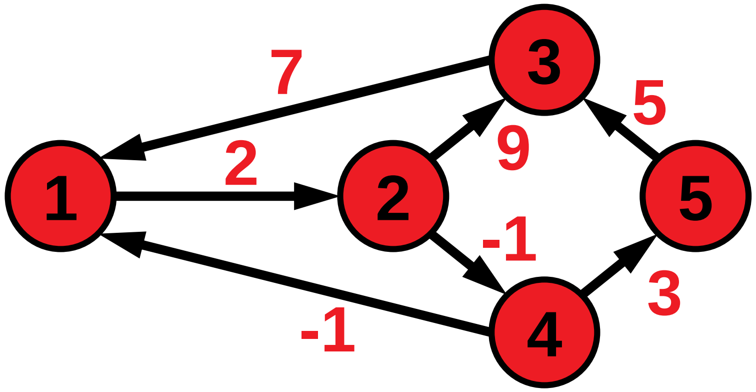
allow {1,2,3,4} as intermediate nodes

$\longrightarrow$

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

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

Example

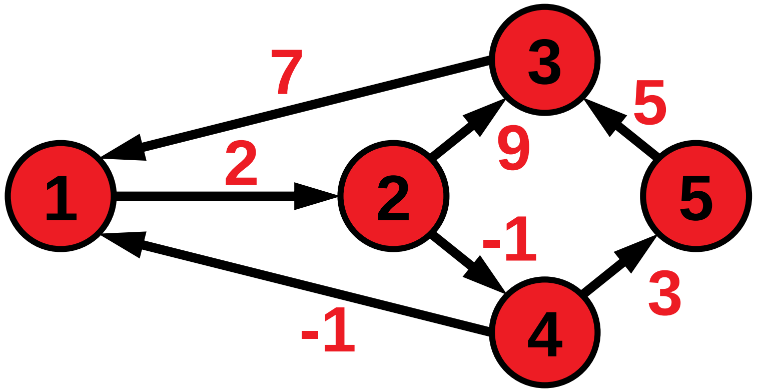


allow {1,2,3,4} as intermediate nodes

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

k=4	1	2	3	4	5
1				1	
2				-1	
3				8	
4	-1	1	10	0	3
5				13	

Example

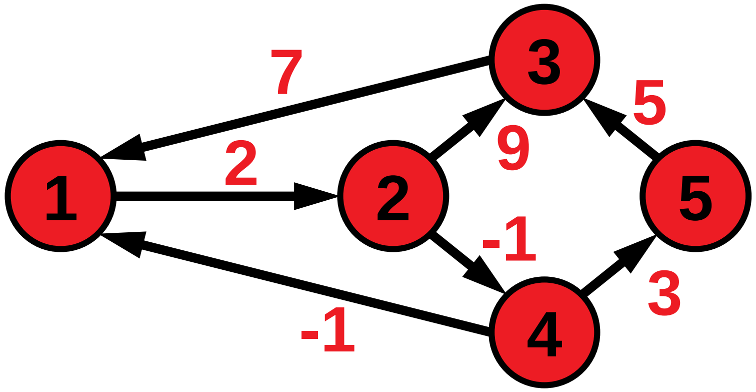


allow {1,2,3,4} as intermediate nodes

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

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

Example

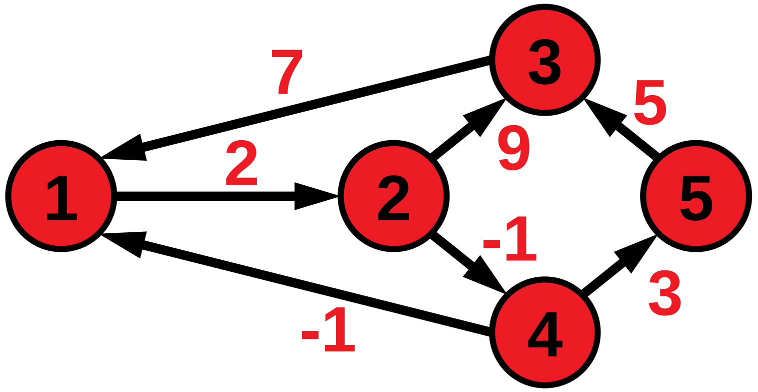

$$\begin{tikzcd}
 & & 3 \arrow[lll, "7"'] & \\
1 \arrow[rr, "2"] & & 2 \arrow[u, "9"] \arrow[d, "-1"] & & 5 \arrow[ull, "5"'] \\
 & & 4 \arrow[ull, "-1"] \arrow[urr, "3"'] &
\end{tikzcd}$$

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
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	9	1	4
2	-2	0	7	-1	2
3	7	9	16	8	11
4	-1	1	8	0	3
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

  dist(i,j) = min { dist(i,j), dist(i,k) + 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
- $P^k_{i,j}$ = predecessor of node j on path from i to j when only vertices 1, ..., k are allowed as intermediate vertices
- no extra costs (asymptotically)

$$P^0_{i,j} = \begin{cases} 0 & \text{if } i = j \text{ or } d_{i,j} = \infty \\ i & \text{in all other cases} \end{cases}$$

$$P^k_{i,j} = \begin{cases} P^{k-1}_{i,j} & \text{if } \mathrm{dist}(i,j) \leq \mathrm{dist}(i,k) + \mathrm{dist}(k,j) \\ P^{k-1}_{k,j} & \text{if } \mathrm{dist}(i,j) > \mathrm{dist}(i,k) + \mathrm{dist}(k,j) \end{cases}$$

Exercise:
The Knapsack Problem and Dynamic Programming

`http://researchers.lille.inria.fr/~brockhof/optimizationSaclay/`

Conclusions

I hope it became clear...

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