Optimization for Machine Learning

Lecture 5: Constraints, Discrete Optimization I

December 1, 2022

TC2 - Optimisation

Université Paris-Saclay

Anne Auger and Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

Date		Topic
Thu, 3.11.2022	DB	Introduction
Thu, 10.11.2022	AA	Continuous Optimization I: differentiability, gradients, convexity, optimality conditions
Thu, 17.11.2022	AA	Continuous Optimization II: constrained optimization, gradient-based algorithms, stochastic gradient
Thu, 24.11.2022	AA	Continuous Optimization III: stochastic algorithms, derivative-free optimization written test / « contrôle continue »
Thu, 1.12.2022	DB	Constrained optimization, Discrete Optimization I: graph theory, greedy algorithms
Thu, 8.12.2022	DB	Discrete Optimization II: dynamic programming, branch&bound
Thu 15.12.2022	DB	Written exam
		classes from 13h30 – 16h45 (2nd break at end)

Constrained Optimization

Small exercises on whiteboard

Equality Constraint

Objective:

Generalize the necessary condition of $\nabla f(x) = 0$ at the optima of f *when f is in $\mathcal{C}^1$, i.e. is differentiable and its differential is continuous*

Theorem:

Be $f: \mathbb{R}^n \to \mathbb{R}$, $g: \mathbb{R}^n \to \mathbb{R}$ in $\mathcal{C}^1$.

Let $a \in \mathbb{R}^n$ satisfy

$$\begin{cases} f(a) = \min \{f(x) \mid x \in \mathbb{R}^n, g(x) = 0\} \\ g(a) = 0 \end{cases}$$

i.e. a is optimum of the problem

If $\nabla g(a) \neq 0$, then there exists a constant $\lambda \in \mathbb{R}$ called *Lagrange multiplier*, such that

$$\underbrace{\nabla f(a) + \lambda \nabla g(a) = 0}_{\text{i.e. gradients of } f \text{ and } g \text{ in } a \text{ are colinear}} \quad \text{Euler} - \text{Lagrange equation}$$

Geometrical Interpretation Using an Example

Exercise:

Consider the problem

$$\inf \ \{ f(x,y) \mid (x,y) \in \mathbb{R}^2, g(x,y) = 0\}$$

$f(x,y) = y - x^2 \qquad g(x,y) = x^2 + y^2 - 1 = 0$

1) Plot the level sets of f, plot $g = 0$
2) Compute ∇f and ∇g
3) Find the solutions with $\nabla f + \lambda \nabla g = 0$

   equation solving with 3 unknowns (x, y, λ)
4) Plot the solutions of 3) on top of the level set graph of 1)

Interpretation of Euler-Lagrange Equation

Intuitive way to retrieve the Euler-Lagrange equation:

- In a local minimum a of a constrained problem, the hypersurfaces (or level sets) $f = f(a)$ and $g = 0$ are necessarily tangent (otherwise we could decrease f by moving along $g = 0$).
- Since the gradients $\nabla f(a)$ and $\nabla g(a)$ are orthogonal to the level sets $f = f(a)$ and $g = 0$, it follows that $\nabla f(a)$ and $\nabla g(a)$ are colinear.

Generalization to More than One Constraint

Theorem

- Assume $f: \mathbb{R}^n \to \mathbb{R}$ and $g_k: \mathbb{R}^n \to \mathbb{R}$ $(1 \leq k \leq p)$ are $\mathcal{C}^1$.
- Let a be such that

$$\begin{cases} f(a) = \min \{f(x) \mid x \in \mathbb{R}^n, \quad g_k(x) = 0, \quad 1 \leq k \leq p\} \\ \qquad\qquad g_k(a) = 0 \ \text{ for all } 1 \leq k \leq p \end{cases}$$

- If $\left(\nabla g_k(a)\right)_{1 \leq k \leq p}$ are linearly independent, then there exist p real constants $(\lambda_k)_{1 \leq k \leq p}$ such that

$$\nabla f(a) + \sum_{k=1}^{p} \lambda_k \nabla g_k(a) = 0$$

Lagrange multiplier

again: a does not need to be global but local minimum

The Lagrangian

- Define the Lagrangian on $\mathbb{R}^n \times \mathbb{R}^p$ as

$$\mathcal{L}(x, \{\lambda_k\}) = f(x) + \sum_{k=1}^{p} \lambda_k g_k(x)$$

- To find optimal solutions, we can solve the optimality system

$$\begin{cases} \text{Find } (x, \{\lambda_k\}) \in \mathbb{R}^n \times \mathbb{R}^p \text{ such that } \nabla f(x) + \sum_{k=1}^{p} \lambda_k \nabla g_k(x) = 0 \\ g_k(x) = 0 \ \text{ for all } 1 \leq k \leq p \end{cases}$$

$$\Leftrightarrow \begin{cases} \text{Find } (x, \{\lambda_k\}) \in \mathbb{R}^n \times \mathbb{R}^p \text{ such that } \nabla_x \mathcal{L}(x, \{\lambda_k\}) = 0 \\ \nabla_{\lambda_k} \mathcal{L}(x, \{\lambda_k\})(x) = 0 \ \text{ for all } 1 \leq k \leq p \end{cases}$$

Inequality Constraint: Definitions

Let $\mathcal{U} = \{x \in \mathbb{R}^n \mid g_k(x) = 0 \text{ (for } k \in E), \; g_k(x) \leq 0 \text{ (for } k \in I)\}$.

Definition:

The points in $\mathbb{R}^n$ that satisfy the constraints are also called *feasible* points.

Definition:

Let $a \in \mathcal{U}$, we say that the constraint $g_k(x) \leq 0$ (for $k \in I$) is *active* in a if $g_k(a) = 0$.

Inequality Constraint: Karush-Kuhn-Tucker Theorem

Theorem (Karush-Kuhn-Tucker, KKT):

Let $f: \mathbb{R}^n \to \mathbb{R}$, $g_k: \mathbb{R}^n \to \mathbb{R}$, all $\mathcal{C}^1$

Furthermore, let $a \in \mathbb{R}^n$ satisfy

$$\begin{cases} f(a) = \min(f(x) \mid x \in \mathbb{R}^n, g_k(x) = 0 \ (\text{for } k \in E), g_k(x) \leq 0 \ (\text{for } k \in \mathrm{I}) \\ g_k(a) = 0 \ (\text{for } k \in E) \\ g_k(a) \leq 0 \ (\text{for } k \in I) \end{cases}$$

also works again for a being a local minimum

Let I_a^0 be the set of constraints that are active in a. Assume that $\left(\nabla g_k(a)\right)_{k \in E \cup I_a^0}$ are linearly independent.

Then there exist $(\lambda_k)_{1 \leq k \leq p}$ that satisfy

$$\begin{cases} \nabla f(a) + \sum_{k=1}^{p} \lambda_k \nabla g_k(a) = 0 \\ g_k(a) = 0 \ (\text{for } k \in E) \\ g_k(a) \leq 0 \ (\text{for } k \in I) \\ \lambda_k \geq 0 \ (\text{for } k \in I_a^0) \\ \lambda_k g_k(a) = 0 \ (\text{for } k \in E \cup I) \end{cases}$$

Inequality Constraint: Karush-Kuhn-Tucker Theorem

Theorem (Karush-Kuhn-Tucker, KKT):

Let $f: \mathbb{R}^n \to \mathbb{R}$, $g_k: \mathbb{R}^n \to \mathbb{R}$, all $\mathcal{C}^1$

Furthermore, let $a \in \mathbb{R}^n$ satisfy

$$\begin{cases} f(a) = \min(f(x) \mid x \in \mathbb{R}^n, g_k(x) = 0 \text{ (for } k \in E), g_k(x) \leq 0 \text{ (for } k \in \mathrm{I}) \\ g_k(a) = 0 \text{ (for } k \in E) \\ g_k(a) \leq 0 \text{ (for } k \in I) \end{cases}$$

Let I_a^0 be the set of constraints that are active in a. Assume that $\left(\nabla g_k(a)\right)_{k \in E \cup I_a^0}$ are linearly independent.

Then there exist $(\lambda_k)_{1 \leq k \leq p}$ that satisfy

$$\begin{cases} \nabla f(a) + \sum_{k=1}^{p} \lambda_k \nabla g_k(a) = 0 \\ g_k(a) = 0 \text{ (for } k \in E) \\ g_k(a) \leq 0 \text{ (for } k \in I) \\ \lambda_k \geq 0 \text{ (for } k \in I_a^0) \\ \lambda_k g_k(a) = 0 \text{ (for } k \in E \cup I) \end{cases}$$

either active constraint or $\lambda_k = 0$

Discrete Optimization

Discrete Optimization

Context discrete optimization:

- discrete variables
- or optimization over discrete structures (e.g. graphs)
- search space often finite, but typically too large for enumeration
- → need for smart algorithms

Algorithms for discrete problems:

- typically problem-specific
- but some general concepts are repeatedly used:
  - greedy algorithms
  - [branch and bound]
  - dynamic programming
  - randomized search heuristics

before 2 excursions:
the O-notation
& graph theory

Motivation for this Part:

  - get an idea of the most common algorithm design principles

Excursion: The O-Notation

Excursion: The O-Notation

Motivation:

- we often want to characterize how quickly a function f(x) grows asymptotically
- e.g. when we say an algorithm takes quadratically many steps (in the input size) to find the optimum of a problem with n (binary) variables, it is most likely not exactly n^2, but maybe n^2+1 or $(n+1)^2$

Big-O Notation

should be known, here mainly restating the definition:

Definition 1 We write $f(x) = O(g(x))$ iff there exists a constant $c > 0$ and an $x_0 > 0$ such that $|f(x)| \leq c \cdot g(x)$ holds for all $x > x_0$

we also view O(g(x)) as a set of functions growing at most as quick as g(x) and write $f(x) \in O(g(x))$

Big-O: Examples

- $f(x) + c = O(f(x))$ [if f(x) does not go to zero for x to infinity]
- $c \cdot f(x) = O(f(x))$
- $f(x) \cdot g(x) = O(f(x) \cdot g(x))$
- $3n^4 + n^2 - 7 = O(n^4)$

Intuition of the Big-O:

- if $f(x) = O(g(x))$ then g(x) gives an upper bound (asymptotically) for f
  excluding constants and lower order terms
- With Big-O, you should have '≤' in mind
- An algorithm that solves a problem in polynomial time is "efficient"
- An algorithm that solves a problem in exponential time is not
- But be aware:
  In practice, often the line between efficient and non-efficient lies around $n \log n$ or even n (or even $\log n$ in the big data context) and the constants **do** matter!!!

Excursion: The O-Notation

Further definitions to generalize from '≤' to '≥' and '=':

- $f(x) = \Omega(g(x))$ if $g(x) = O(f(x))$
- $f(x) = \Theta(g(x))$ if $f(x) = O(g(x))$ and $g(x) = O(f(x))$

Note: extensions to '<' and '>' exist as well, but are not needed here.

Example:

- Algo A solves problem P in time $O(n)$
- Algo B solves problem P in time $O(n^2)$
- which one is faster?

only proving upper bounds to compare algorithms is not sufficient!

Excursion: The O-Notation

Further definitions to generalize from '≤' to '≥' and '=':

- $f(x) = \Omega(g(x))$ if $g(x) = O(f(x))$
- $f(x) = \Theta(g(x))$ if $f(x) = O(g(x))$ and $g(x) = O(f(x))$

Note: extensions to '<' and '>' exist as well, but are not needed here.

Example:

- Algo A solves problem P in time $O(n)$
- Algo B solves problem P in time $\cancel{O(n^2)}$ $\Omega(n^2)$
- which one is faster?

only proving upper bounds to compare algorithms is not sufficient!

Exercise O-Notation

❶ Please order the following functions in terms of their asymptotic behavior (from smallest to largest):

- $\exp(n^2)$
- $\log n$
- $\ln n / \ln \ln n$
- n
- $n \log n$
- $\exp(n)$
- $\ln n!$

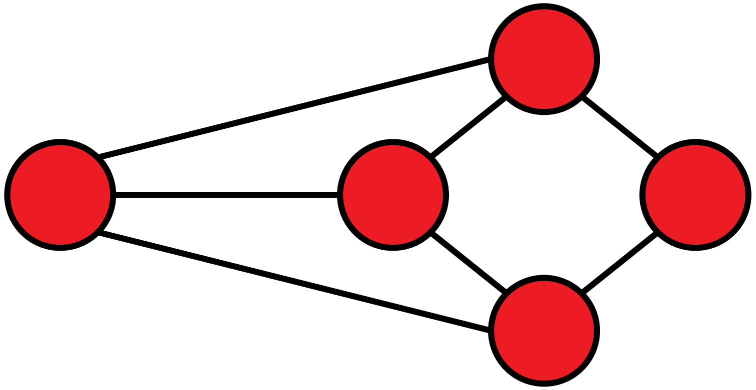
❷ Pick one pair of runtimes and give a formal proof for the relation.

Excursion:
Basic Concepts of Graph Theory

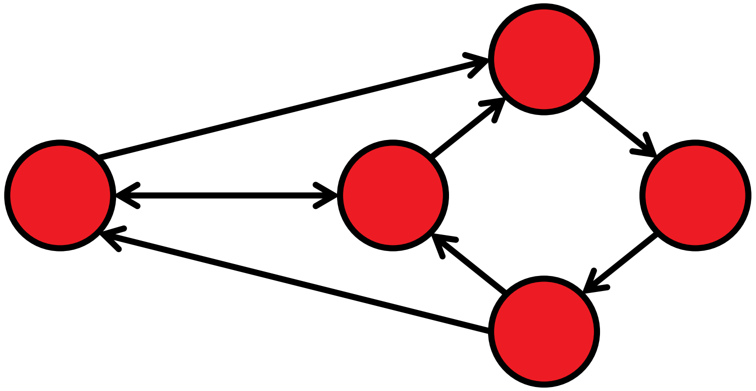
[following for example http://math.tut.fi/~ruohonen/GT_English.pdf]

Graphs

Definition 1 *An undirected graph G is a tupel $G = (V, E)$ of edges $e = \{u, v\} \in E$ over the vertex set V (i.e., $u, v \in V$).*

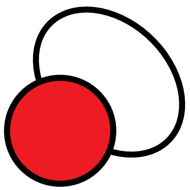


- vertices = nodes
- edges = lines
- Note: edges cover two *unordered* vertices (*undirected* graph)
  - if they are *ordered*, we call G a *directed* graph



Graphs: Basic Definitions

- G is called *empty* if E empty
- u and v are *end vertices* of an edge {u,v}
- Edges are *adjacent* if they share an end vertex
- Vertices u and v are *adjacent* if {u,v} is in E

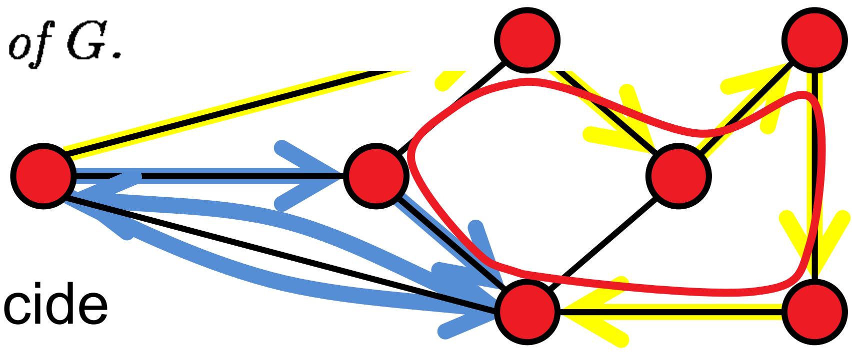


a loop

Definition 1 *A* walk *in a graph* $G = (V, E)$ *is a sequence*

$$v_{i_0}, e_{i_1} = (v_{i_0}, v_{i_1}), v_{i_1}, e_{i_2} = (v_{i_1}, v_{i_2}), \dots, e_{i_k}, v_{i_k},$$

alternating vertices and adjacent edges of G.



A walk is

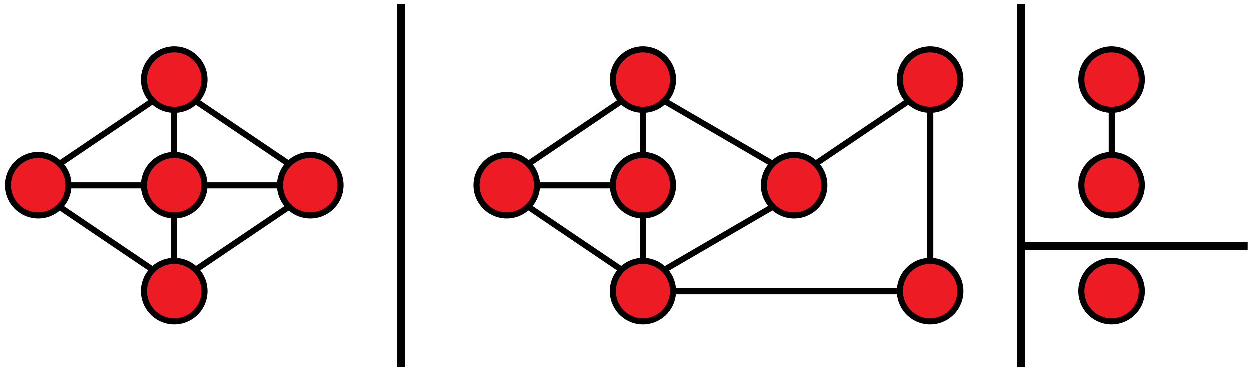
- *closed* if first and last node coincide
- a *trail* if each edge traversed at most once
- a *path* if each vertex is visited at most once

a closed path is called a *circuit* or *cycle*

Graphs: Connectedness

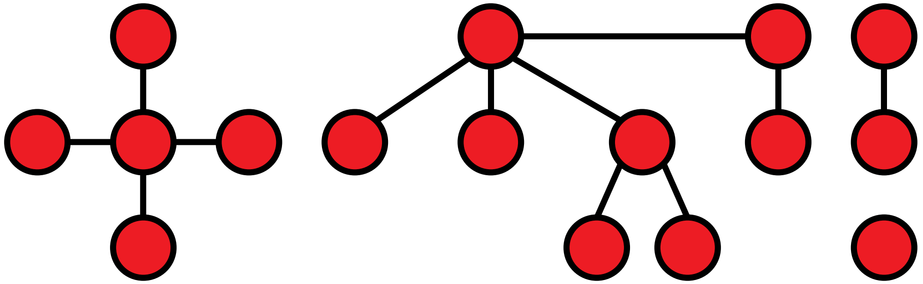
- Two vertices are called *connected* if there is a walk between them in G
- If all vertex pairs in G are connected, G is called connected

- The *connected components* of G are the (maximal) subgraphs which are connected.

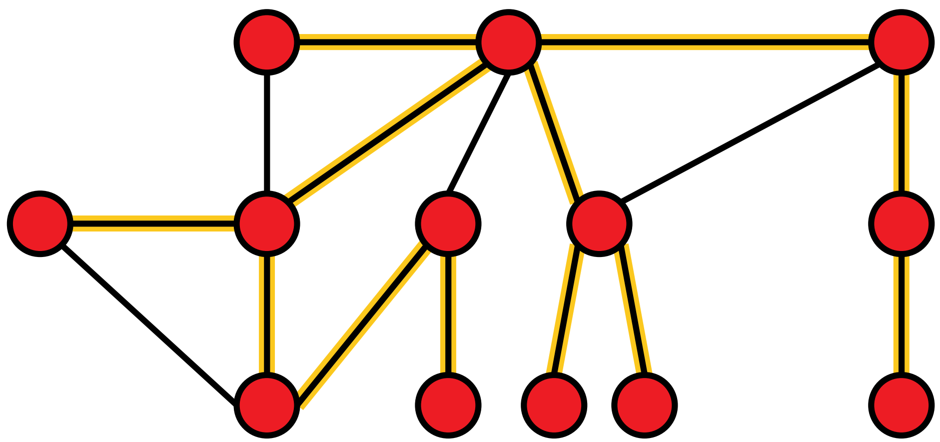


Trees and Forests

- A *forest* is a cycle-free graph
- A *tree* is a connected forest



A *spanning tree* of a connected graph G is a tree in G which contains all vertices of G



Greedy Algorithms

Greedy Algorithms

From Wikipedia:

"A *greedy algorithm* is an algorithm that follows the problem solving *heuristic* of making the locally optimal choice at each stage with the hope of finding a global optimum."

- Note: typically greedy algorithms do not find the global optimum

Lecture Outline Greedy Algorithms

What we will see:

❶ Example 1: Money Change problem

❷ Example 2: Minimal Spanning Trees (MST) and the algorithm of Kruskal

Example 1: Money Change

Change-making problem

- Given n coins of distinct values $w_1=1$, w_2, ..., w_n and a total change W (where w_1, ..., w_n, and W are integers).
- Minimize the total amount of coins Σx_i such that $\Sigma w_i x_i = W$ and where x_i is the number of times, coin i is given back as change.

Greedy Algorithm

Unless total change not reached:

add the largest coin which is not larger than the remaining amount to the change

Note: only optimal for standard coin sets, not for arbitrary ones!

Related Problem:

finishing darts (from 501 to 0 with 9 darts)

Example 2: Minimal Spanning Trees (MST)

Outline:

- problem definition
- Kruskal's algorithm
  - including correctness proofs and analysis of running time

MST: Problem Definition

Reminder: A *spanning tree* of a connected graph G is a tree in G which contains all vertices of G

Minimum Spanning Tree Problem (MST):

Given a (connected) graph $G = (V, E)$ with edge weights w_i for each edge e_i. Find a spanning tree T that minimizes the weights of the contained edges, i.e. where

$$\sum_{e_i \in T} w_i$$

is minimized.

Kruskal's Algorithm

Idea (Kruskal, 1956):

- create (minimal) spanning tree from the bottom up
- start with all nodes separate/unconnected
- in each step, connect a so-far unconnected node to another one
  - without creating a cycle
  - until we have found our (minimal) spanning tree
  - …and in a way that creates a minimal spanning tree in the end

Kruskal's Algorithm

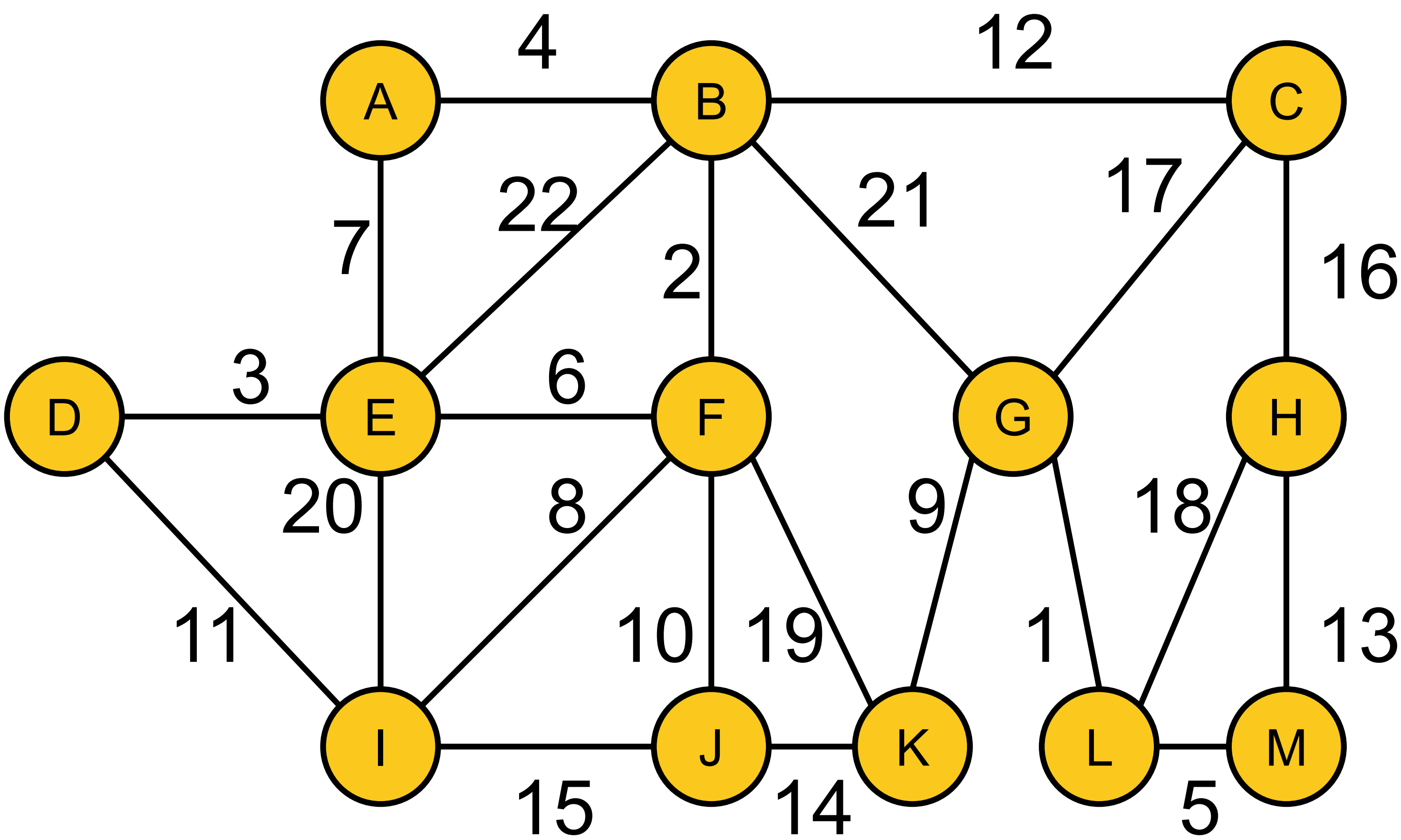
Algorithm, see [1]

- Create forest F = (V,{}) with n components and no edge
- Put sorted edges (such that w.l.o.g. $w_1 \leq w_2 \leq ... \leq w_{|E|}$) into S
- While S non-empty and F not spanning:
  - delete cheapest edge from S
  - add it to F if no cycle is introduced

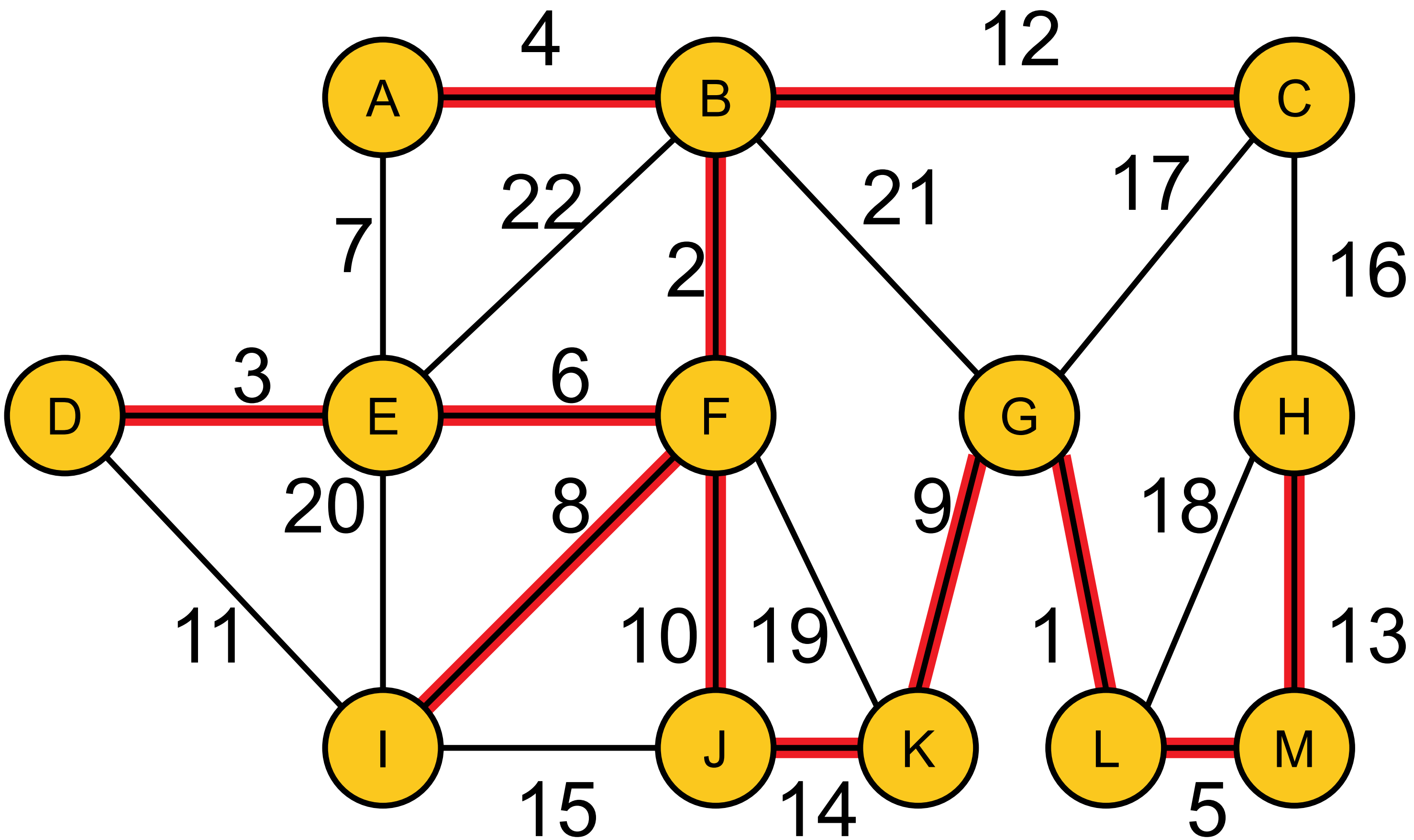
Where is the greedy part?

[1] Kruskal, J. B. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem". *Proceedings of the American Mathematical Society* **7**: 48–50. doi:10.1090/S0002-9939-1956-0078686-7

Kruskal's Algorithm: Example



Kruskal’s Algorithm: Example



Kruskal's Algorithm: Runtime Considerations

First question: how to implement the algorithm?

- sorting of edges needs O(|E| log |E|)

Algorithm

Create forest F = (V,{}) with n components and no edge

Put sorted edges (such that w.l.o.g. $w_1 \leq w_2 \leq ... \leq w_{|E|}$) into S

While S non-empty and F not spanning:

delete cheapest edge from S

add it to F if no cycle is introduced

simple

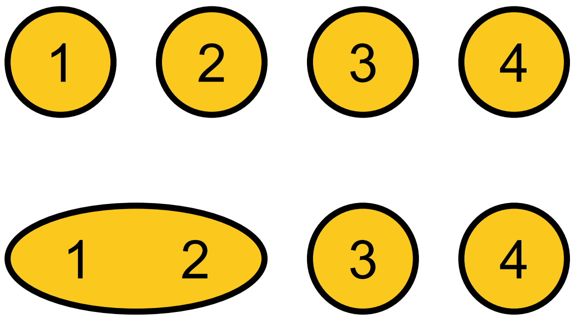
?

forest implementation:
Disjoint-set data structure

Disjoint-set Data Structure ("Union&Find")

Data structure: ground set 1...N grouped to disjoint sets

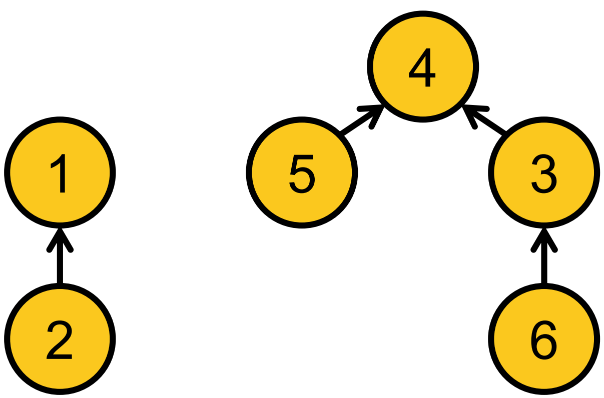
Operations:



- FIND(i): to which set ("tree") does i belong?
- UNION(i,j): union the sets of i and j!
  ("join the two trees of i and j")

Implemented as trees:

- UNION(T1, T2): hang root node of smaller tree under root node of larger tree (constant time), thus
- FIND(u): traverse tree from u to root (to return a representative of u's set) takes logarithmic time in total number of nodes



Implementation of Kruskal's Algorithm

Algorithm, rewritten with UNION-FIND:

- Create initial disjoint-set data structure, i.e. for each vertex v_i, store v_i as representative of its set
- Create empty forest F = (V, {})
- Sort edges into $e_1, \ldots, e_{|E|}$ such that $w_1 < w_2 < \ldots < w_{|E|}$
- for each edge e_i={u,v} starting from i=1:
  - if FIND(u) ≠ FIND(v): # no cycle introduced
    - $F = F \cup \{\{u,v\}\}$
    - UNION(u,v)
  - If UNION-FIND data structure has only one component:
    - return F

Back to Runtime Considerations

- Sorting of edges needs O(|E| log |E|)
- forest: **Disjoint-set data structure**
  - initialization: O(|V|)
  - O(log |V|) to find out whether the minimum-cost edge {u,v} connects two sets (no cycle induced) or is within a set (cycle would be induced)
  - 2x FIND + potential UNION needs to be done O(|E|) times
  - total O(|E| log |V|)
- Overall: O(|E| log |E|)

Kruskal's Algorithm: Proof of Correctness

Two parts needed:

❶ Algo always produces a spanning tree

final F contains no cycle and is connected by definition ✓

❷ Algo always produces a *minimum* spanning tree

- argument by induction
- P: If *F* is forest at a given stage of the algorithm, then there is some minimum spanning tree that contains *F*.
- clearly true for F = (V, {})
- assume that P holds when new edge e is added to F and be T the MST that contains F
  - if e in T, fine
  - if e not in T: T + e has cycle C with edge f in C but not in F (otherwise e would have introduced a cycle in F)
    - now T – f + e is a tree with same weight as T (since T is a MST and f was not chosen to F)
    - hence T – f + e is MST including T + e (i.e. P holds)

✓

Conclusion Greedy Algorithms I

What we have seen so far:

- two problems where a greedy algorithm was optimal
  - money change
  - minimum spanning tree (Kruskal's algorithm)
- but also: greedy not always optimal
  - for some sets of coins for example

Obvious Question: when is greedy good?

Answer: if the problem is a matroid (no further details here)

From Wikipedia: [...] a matroid is a structure that captures and generalizes the notion of linear independence in vector spaces. There are many equivalent ways to define a matroid, the most significant being in terms of independent sets, bases, circuits, closed sets or flats, closure operators, and rank functions.

Conclusions Greedy Algorithms II

I hope it became clear...

...what a greedy algorithm is

...that it not always results in the optimal solution

...but that it does if and only if the problem is a matroid

Dynamic Programming

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

Note:

the reason why the approach is called "dynamic programming" is historical: at the time of invention by Richard Bellman, no computer "program" existed

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:

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

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

➌ bottom-up solving of the subproblems (i.e. computing their optimal value), starting from the smallest by using the Bellman equality and a table structure to store the optimal values (top-down approach also possible, but less common)

➍ eventually construct the final solution (can be omitted if only the value of an optimal solution is sought)

Lecture Outline Dynamic Programming (DP)

What we will see:

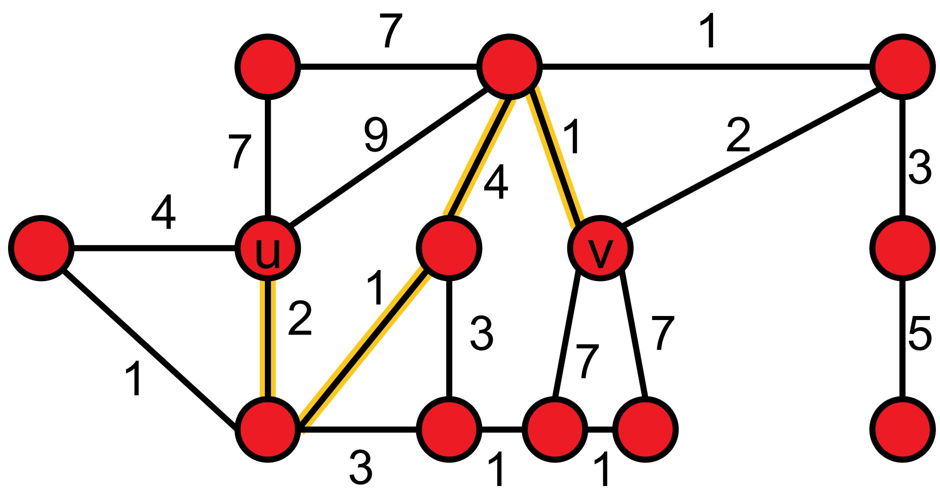
❶ Example 1: The All-Pairs Shortest Path Problem

❷ Example 2: The knapsack problem

Example 1: The 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.



Obvious Applications

Google maps

Autonomous cars

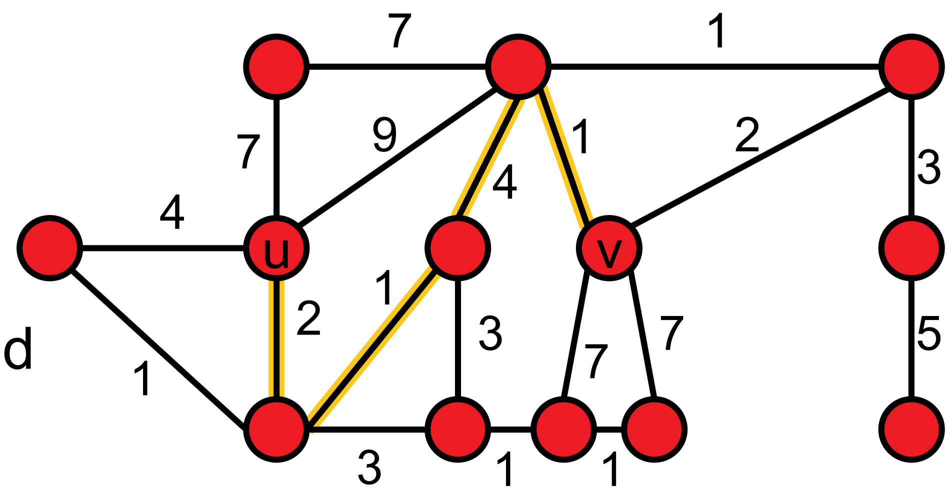
Finding routes for packages in a computer network

...

Example 1: The 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:

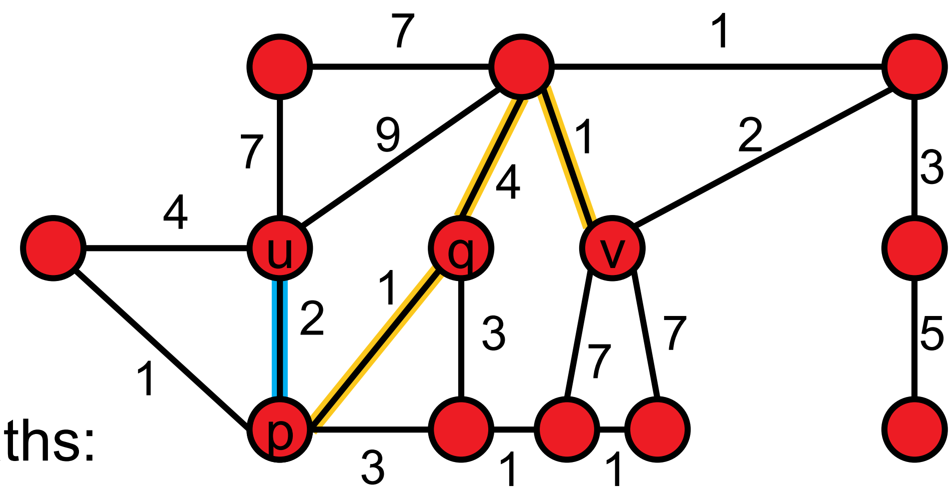
We can often assume that the edge weights are stored in a distance matrix D of dimension |E|x|E| where an entry $D_{i,j}$ gives the weight between nodes i and j and "non-edges" are assigned a value of ∞

Why important? ⇨ determines input size

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.



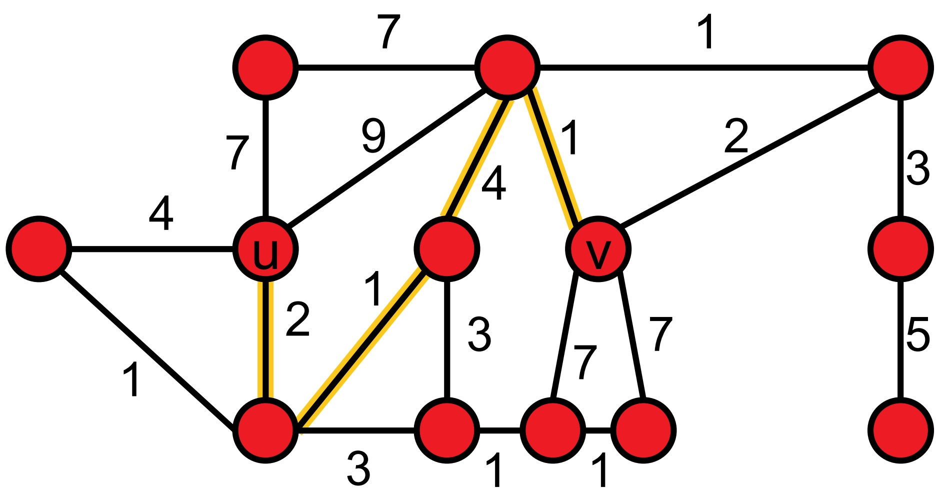
Overlapping Subproblems

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 All Pairs Shortest Paths Problem

All Pairs Shortest Path problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i.
Find the shortest path from each source vertex v to each other target vertex u, i.e., the paths (v, e_1={v, v_1}, v_1, ..., v_k, e_k={v_k,u}, u) such that $w_1 + ... + w_k$ is minimized for all pairs (u,v) in V^2.



The Algorithm of Robert Floyd (1962)

Idea:

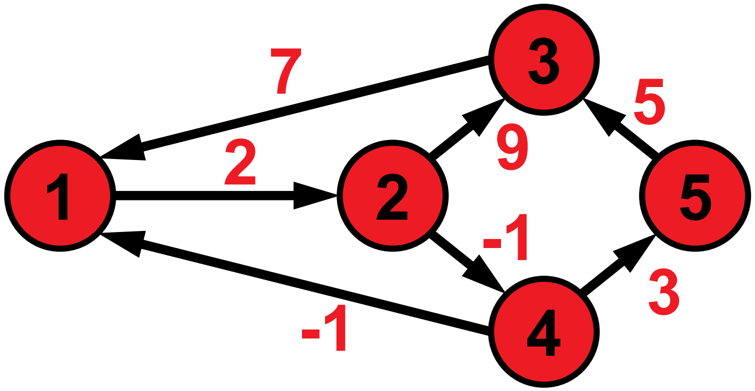
- if we knew that the shortest path between source and target goes through node v, we would be able to construct the optimal path from the shorter paths "source→v" and "v→target"
- subproblem P(k): compute all shortest paths where the intermediate nodes can be chosen from $v_1, \ldots, v_k$

AllPairsShortestPathFloyd(G, D)

- 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. $1 \leq i,j \leq |V|$):
    - $dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$

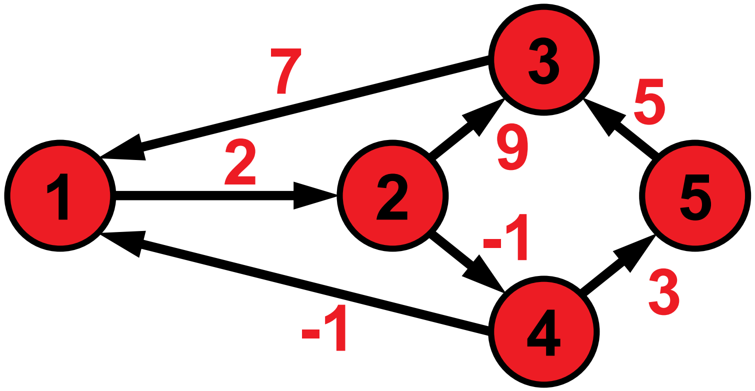
Note: Bernard Roy in 1959 and Stephen Warshall in 1962 essentially proposed the same algorithm independently.

Example



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

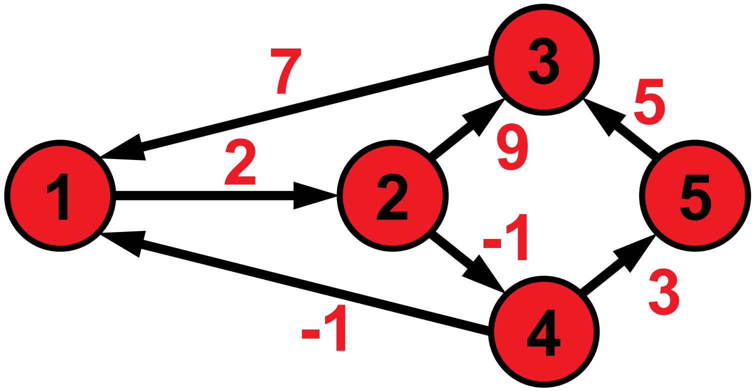
Example



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

Example

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



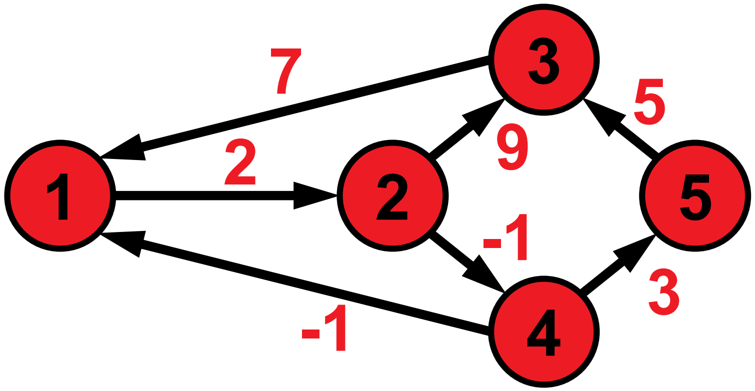
allow 1 as intermediate node

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

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



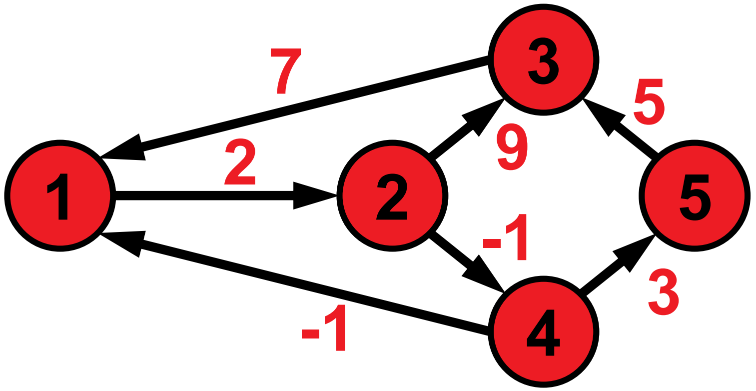
allow 1 as intermediate node

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

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



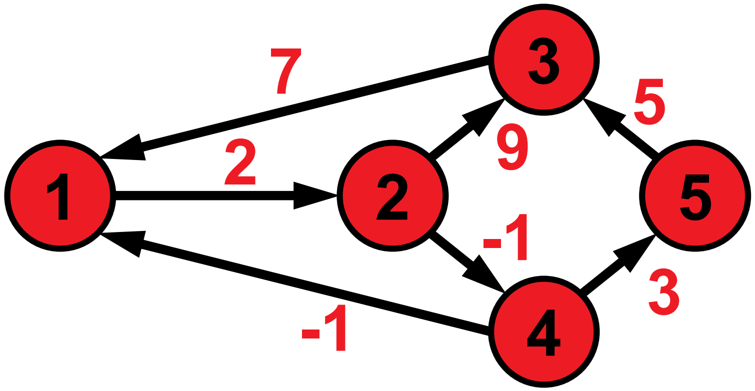
allow 1 as intermediate node

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

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$



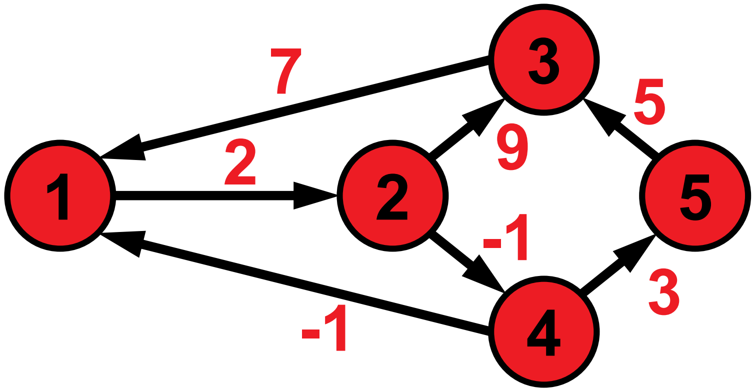
allow 1 as intermediate node

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

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



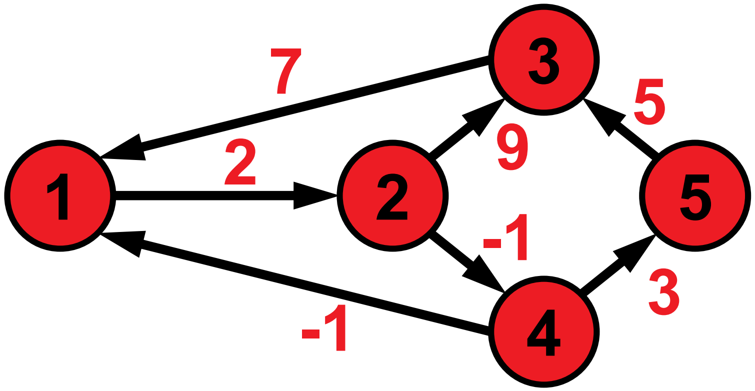
allow 1 as intermediate node

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

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$



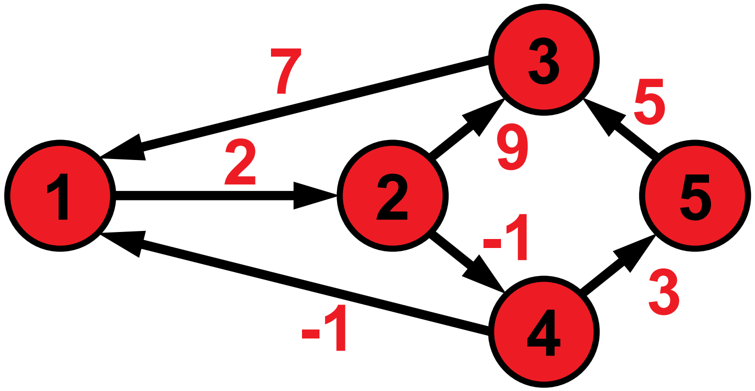
allow 1 & 2 as intermediate nodes

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

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



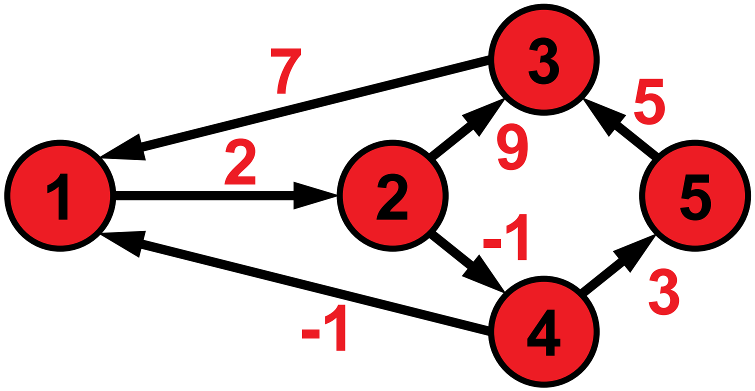
allow 1 & 2 as intermediate nodes

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

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$



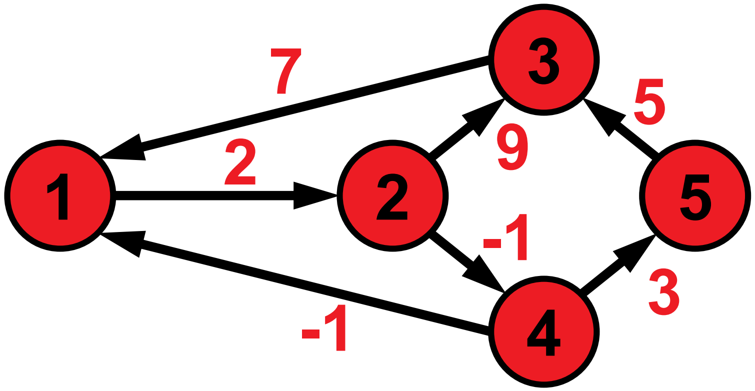
allow 1 & 2 as intermediate nodes

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

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



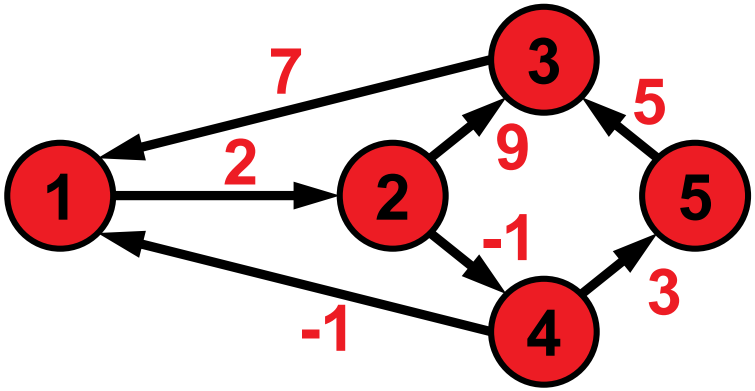
allow {1,2,3} as intermediate nodes

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

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



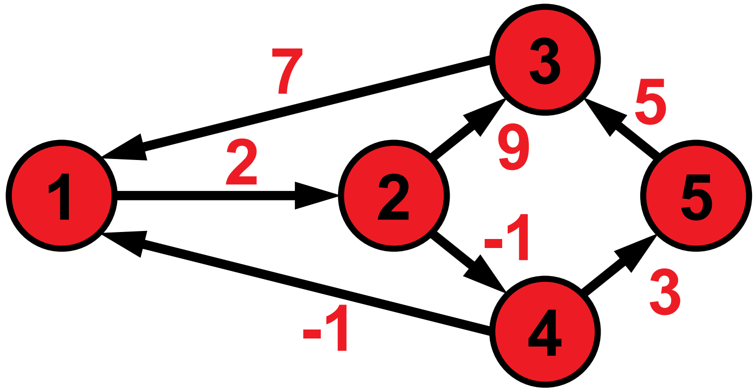
allow {1,2,3} as intermediate nodes

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

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



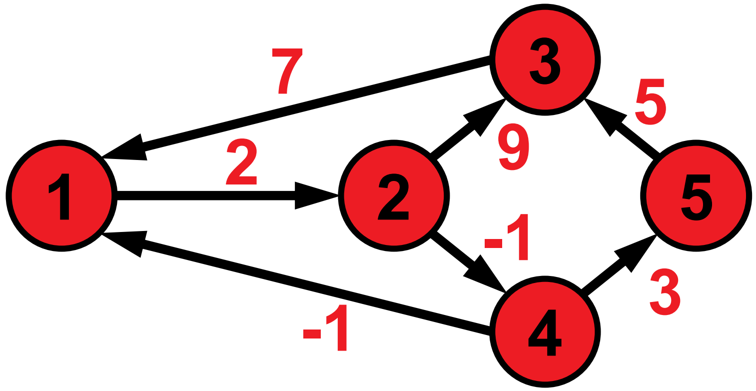
allow {1,2,3} as intermediate nodes

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

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$



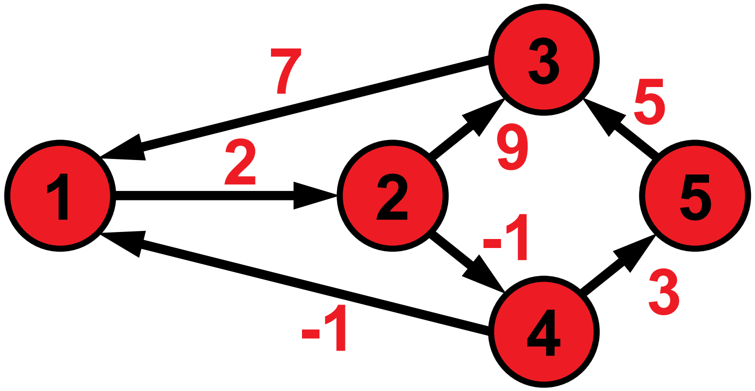
allow {1,2,3,4} as intermediate nodes

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	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

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) }



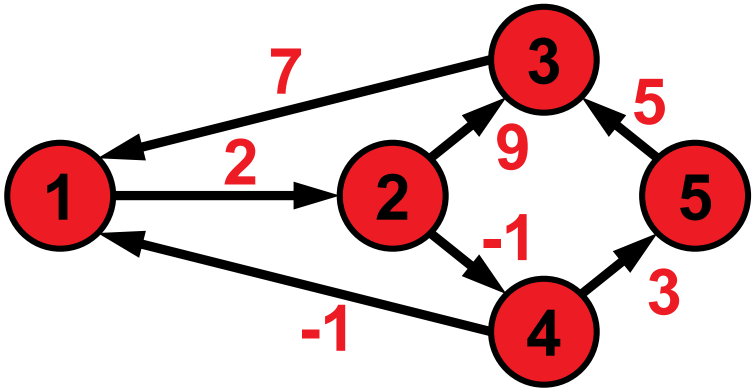
allow {1,2,3,4} as intermediate nodes

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

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) }



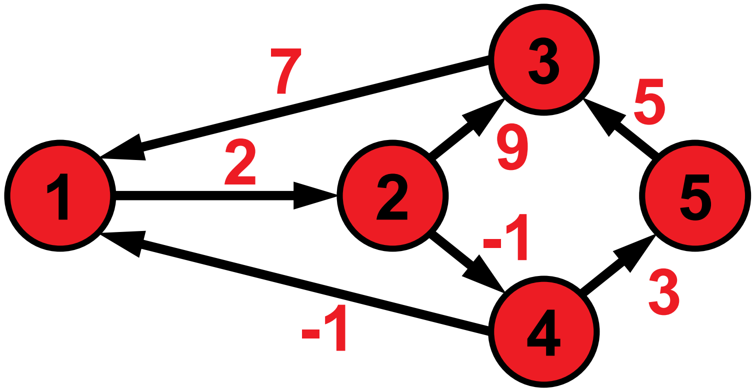
allow {1,2,3,4} as intermediate nodes

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

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$



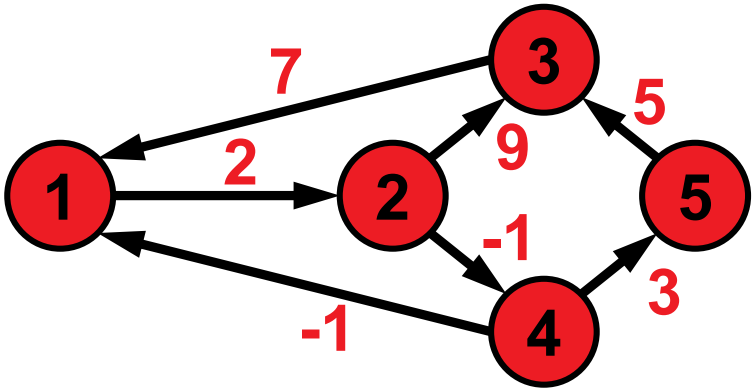
allow all nodes as intermediate nodes

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

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) }



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?

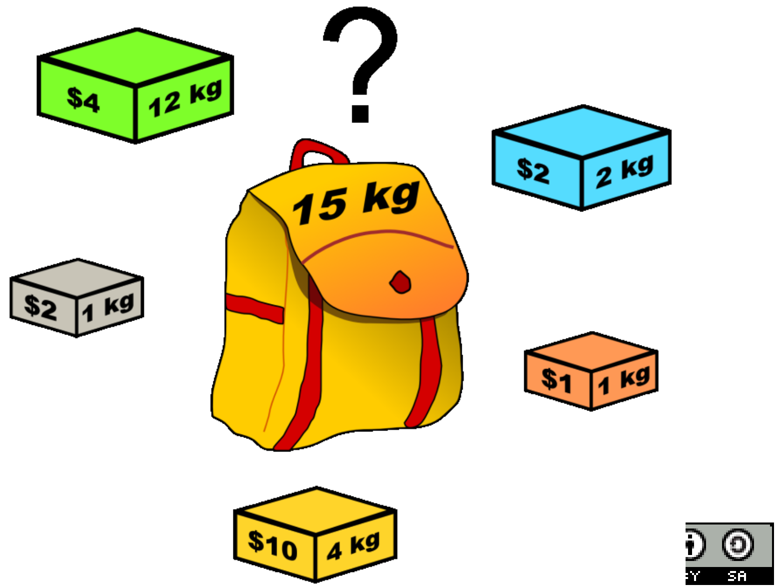
- Construct matrix of predecessors P alongside distance matrix
- $P_{i,j}(k)$ = predecessor of node j on path from i to j (at algo. step k)
- no extra costs (asymptotically)

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

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

Knapsack Problem

[redacted]



Dake

Opt. Substructure and Overlapping Subproblems

Consider the following subproblem:

$P(i,j)$: optimal profit when packing 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 $1 \ldots i-1$:

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

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)...

knapsack weight →

items ↓

P(i,j)	0	1	2	3	...			W-1	W
0									
1					P(i,j)				
2									
...									
n-1									
n									

best achievable profit with items 1...i and a knapsack of size j

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 →

items ↓

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

initialization:
$P(i,j) = 0$ if $i = 0$ 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 →

items ↓

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:

$P(i,j) = 0$ if $i = 0$ 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 →

items ↓

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											

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 →

items ↓

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 →

items ↓

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 →

items ↓

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 →

items ↓

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											

$+p_1 (= 4)$

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 →

items ↓

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											

$+p_1(= 4)$

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 →

items ↓

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											
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 →

items ↓

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:

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

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 →

items ↓

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											

$+p_2(= 10)$

for $i = 1$ to n:

for $j = 1$ to W:

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

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 →

items ↓

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 →

items ↓

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 $\longrightarrow$

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	0											
5	0											

$+p_3(= 3)$

for $i = 1$ to n:

for $j = 1$ to W:

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

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 →

items ↓

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											

$+p_3(=3)$

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 →

items ↓

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											

$+p_3(=3)$

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 →

items ↓

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 →

items ↓

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 →

items ↓

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

$x_1 = 0$

$x_2 = 1$

$x_3 = 0$

$x_4 = 1$

$x_5 = 0$

for $i = 1$ to n:

for $j = 1$ to W:

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

Conclusions

I hope it became clear...

...what the algorithm design ideas of dynamic programming are

...and for which problem types it is supposed to be suitable