Introduction to Optimization
Lectures 6 and 7: Discrete Optimization

October 19, 2018 and October 26, 2018

TC2 - Optimisation

Université Paris-Saclay, Orsay, France

Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

1	Mon, 17.9.2018 Thu, 20.9.2018	Monday's lecture: introduction, example problems, problem types groups defined via wiki everybody went (actively!) through the Getting Started part of github.com/numbbo/coco
2	Fri, 21.9.2018	lecture "Benchmarking", final adjustments of groups everybody can run and postprocess the example experiment (~1h for final questions/help during the lecture)
3	Fri, 28.9.2018	lecture "Introduction to Continuous Optimization"
4	Fri, 5.10.2018	lecture "Gradient-Based Algorithms" + DFO
5	Fri, 12.10.2018	lecture "Stochastic Algorithms", in particular CMA-ES
6	Fri, 19.10.2018	lecture "Discrete Optimization I: graphs, greedy algos, dyn. progr." deadline for submitting data sets
	Wed, 24.10.2018	deadline for paper submission
7	Fri, 26.10.2018	final lecture "Discrete Optimization II: dyn. progr., B&B, heuristics"
	29.10.-2.11.2018	vacation aka learning for the exams
	Thu, 8.11.2018 / Fri, 9.11.2018	oral presentations (individual time slots)
	Fri, 16.11.2018	written exam

All deadlines:
23:59pm Paris time

Course Overview

1	Mon, 17.9.2018 Thu, 20.9.2018	Monday's lecture: introduction, example problems, problem types groups defined via wiki everybody went (actively!) through the Getting Started part of github.com/numbbo/coco
2	Fri, 21.9.2018	lecture "Benchmarking", final adjustments of groups everybody can run and postprocess the example experiment (~1h for final questions/help during the lecture)
3	Fri, 28.9.2018	lecture "Introduction to Continuous Optimization"
4	Fri, 5.10.2018	lecture "Gradient-Based Algorithms" + DFO
5	Fri, 12.10.2018	lecture "Stochastic Algorithms", in particular CMA-ES
6	Fri, 19.10.2018	lecture "Discrete Optimization I: graphs, greedy algos, dyn. progr." deadline for submitting data sets
	Wed, 24.10.2018	deadline for paper submission
7	Fri, 26.10.2018	final lecture "Discrete Optimizatio
	29.10.-2.11.2018	vacation aka learning for the exa
	Thu, 8.11.2018 / Fri, 9.11.2018	oral presentations (individual time slots)
	Fri, 16.11.2018	written exam

Let's check the schedule in the wiki!

All deadlines: 23:59pm Paris time

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

Remark: Coping with Difficult Problems

Exact

- brute-force often too slow
- better strategies such as dynamic programming & branch and bound
- still: often exponential runtime

Approximation Algorithms

- guarantee of low run time
- guarantee of high quality solution
- obstacle: difficult to prove these guarantees

Heuristics

- intuitive algorithms
- guarantee to run in short time
- often no guarantees on solution quality

we will see all 3 kinds of algorithms here...

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 $f(x) \leq c|g(x)|$ 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 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 ~~$O(n^2)$~~ $\Omega(n^2)$
- 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.

Exercise O-Notation (Solution)

Correct ordering:

$\frac{\ln(n)}{\ln(\ln(n))} = O(\log n)$ $\quad \log n = O(n)$ $\quad n = O(n \log n)$

$n \log n = \Theta(\ln(n!))$ $\quad \ln(n!) = O(e^n)$ $\quad e^n = O(e^{n\wedge 2})$

but for example $e^{n\wedge 2} \neq O(e^n)$

One exemplary proof:

$\frac{\ln(n)}{\ln(\ln(n))} = O(\log n)$:

$$\frac{\ln(n)}{\ln(\ln(n))} = \frac{\log(n)}{\log(e)\ln(\ln(n))} \leq \frac{3\log(n)}{\ln(\ln(n))} \leq 3\,|\log(n)|$$

for $n > 1$ for $n > 15$

Exercise O-Notation (Solution)

One additional proof: ln n! = O(n log n)

- Stirling's approximation: $n! \sim \sqrt{2\pi n}\,(n/e)^n$ or even

$$\sqrt{2\pi}\, n^{n+1/2} e^{-n} \leq n! \leq e\, n^{n+1/2} e^{-n}$$

- $\ln n! \leq \ln(e n^{n+\frac{1}{2}} e^{-n}) = 1 + \left(n + \frac{1}{2}\right) \ln n - n$

$$\leq \left(n + \frac{1}{2}\right) \ln n \leq 2n \ln n = 2n \frac{\log n}{\log e} = c \cdot n \log n$$

okay for $c = 2/\log e$ and all $n \in \mathbb{N}$

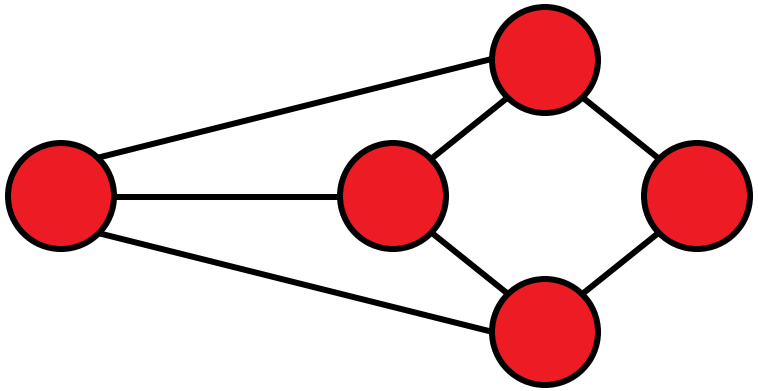
- n ln n = O(ln n!) proven in a similar vein

Excursion:
Basic Concepts of Graph Theory

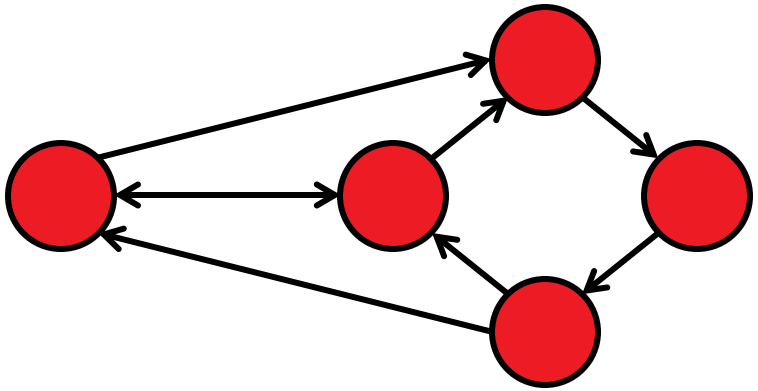
[following for example http://math.tut.fi/~ruohonen/GT_English.pdf]

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., $u, v \in V$).*

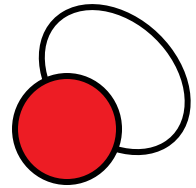


- vertices = nodes
- edges = lines
- Note: edges cover two *unordered* vertices (*undirected* graph)
  - if they are *ordered*, we call G a *directed* graph

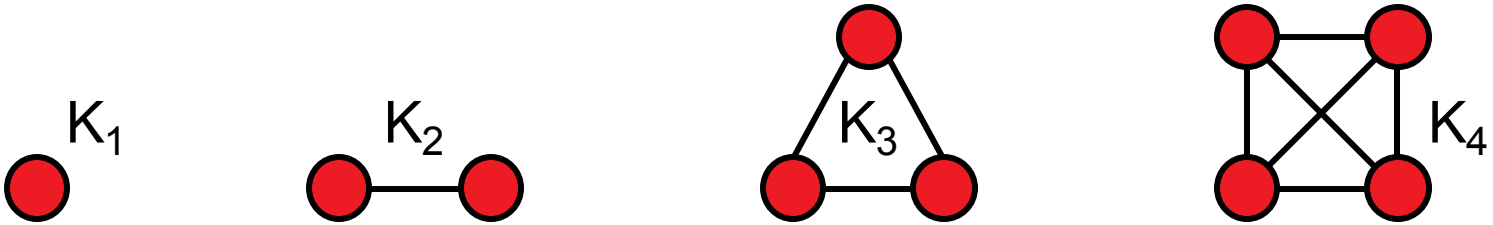


Graphs: Basic Definitions

- 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
- The *degree* of a vertex is the number of times it is an end vertex
- A complete graph contains all possible edges (once):



a loop

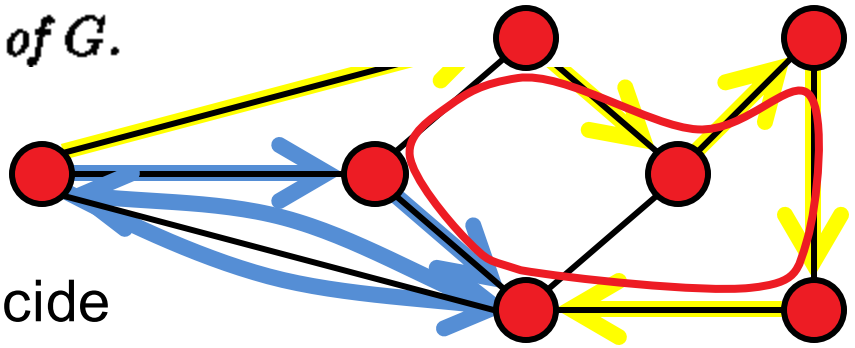


Walks, Paths, and Circuits

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

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

alternating vertices and adjacent edges of G.



A walk is

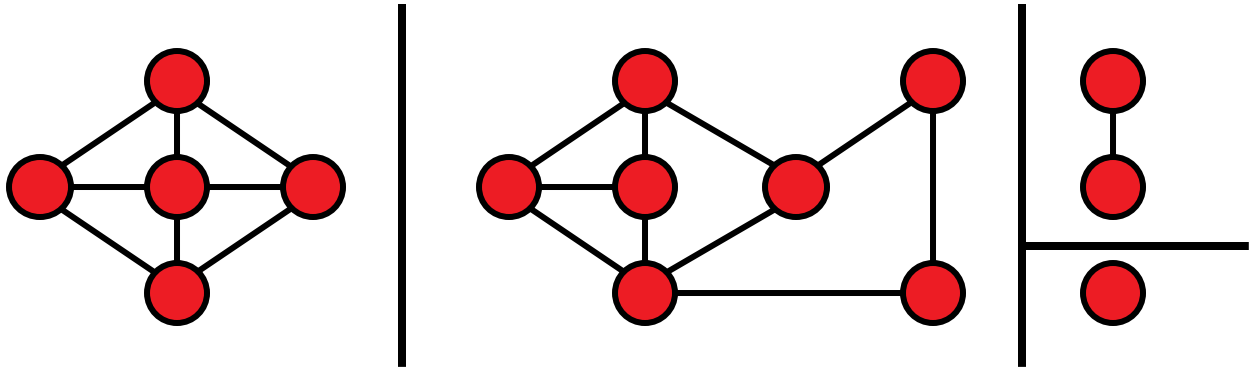
- *closed* if first and last node coincide
- a *trail* if each edge traversed at most once
- a *path* if each vertex is visited at most once

- a closed path is a *circuit* or *cycle*
- a closed path involving all vertices of G is a *Hamiltonian cycle*

Graphs: Connectedness

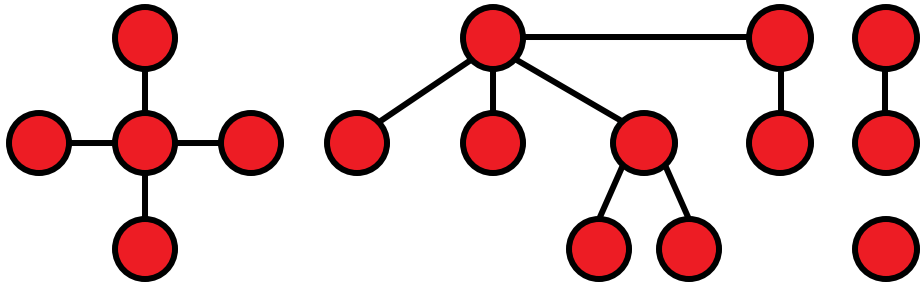
- Two vertices are called *connected* if there is a walk between them in G
- If all vertex pairs in G are connected, G is called connected

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

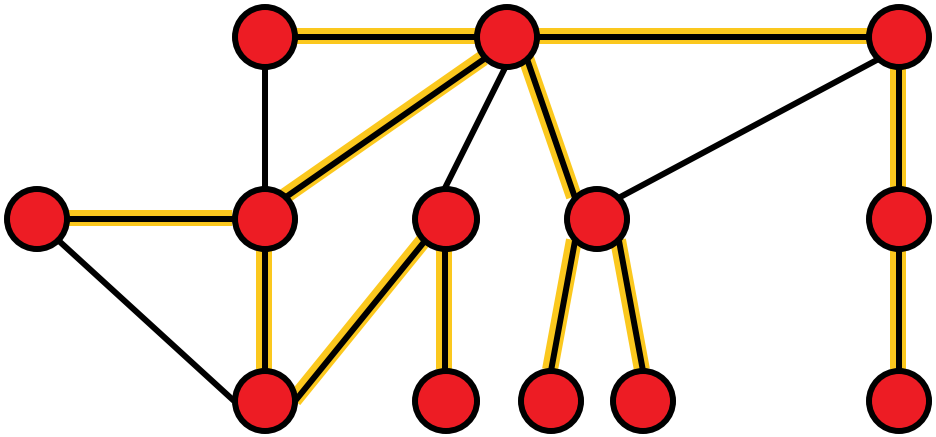


Trees and Forests

- 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
- We will see later when this is the case

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

❸ Example 3: An approximation algorithm for Bin Packing

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
- analysis of its running time
- proof of its correctness

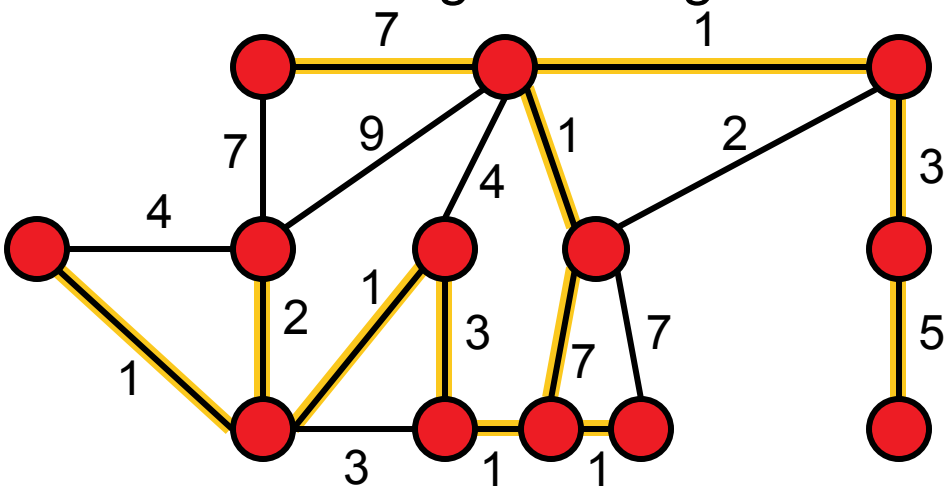
Minimum Spanning Trees (MST)

Minimum Spanning Tree problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i. Find the spanning tree with the smallest weight among all spanning trees.

weight of a spanning tree:

$$w(T) = \sum_{e_i \text{ in } T} w_i$$



w(T) = 33

Applications

Setting up a new wired telecommunication/water supply/electricity network

Constructing minimal delay trees for broadcasting in networks

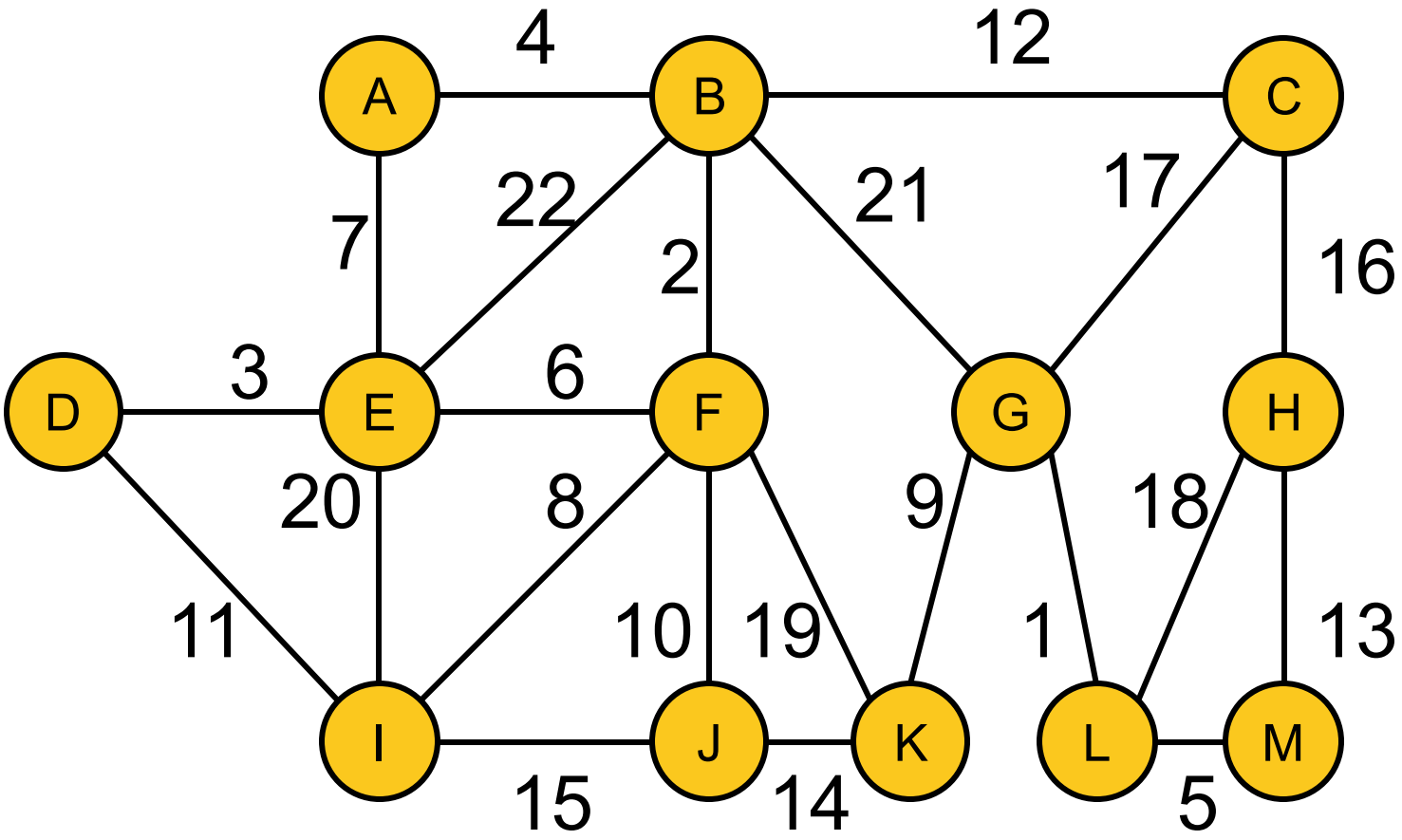
Kruskal's Algorithm: Idea

Algorithm, see [1]

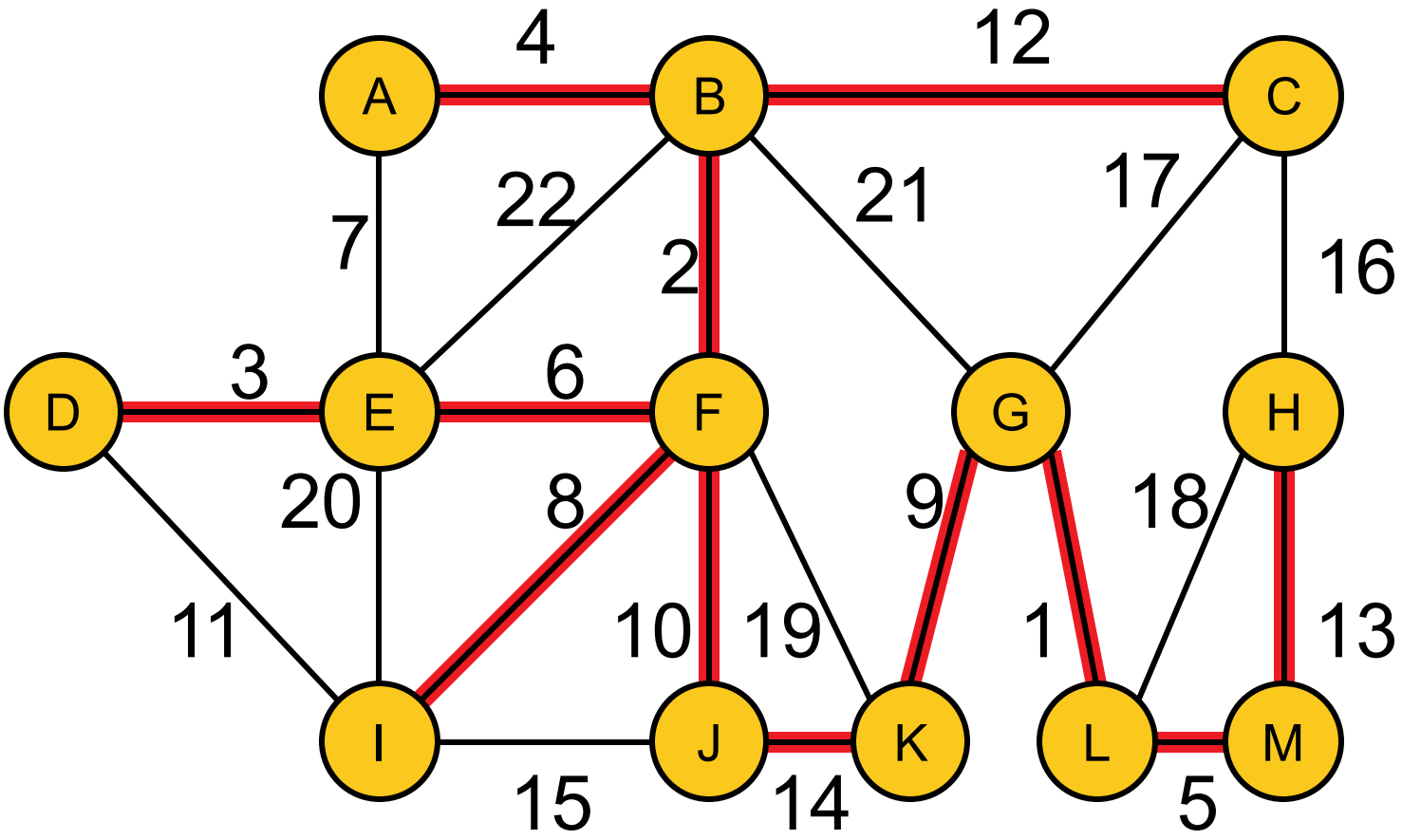
- Create forest F = (V,{}) with n components and no edge
- Put sorted edges (such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$) into set S
- While S non-empty and F not spanning:
  - delete cheapest edge from S
  - add it to F if no cycle is introduced

[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



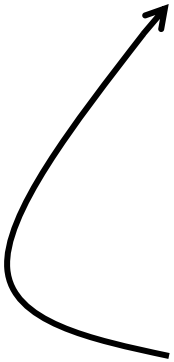
Kruskal's Algorithm: Example



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. $w_1 < w_2 < \ldots < w_{|E|}$) into set S

While S non-empty and F not spanning:

delete cheapest edge from S

add it to F if no cycle is introduced

simple

?

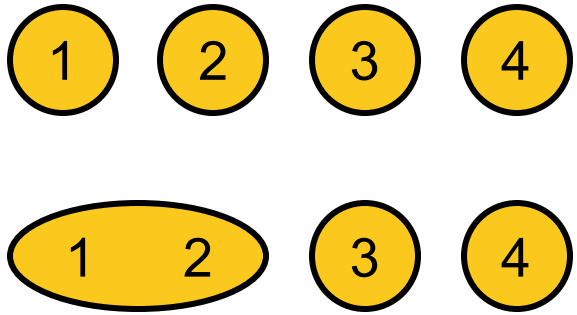
forest implementation:
Disjoint-set data structure

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

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

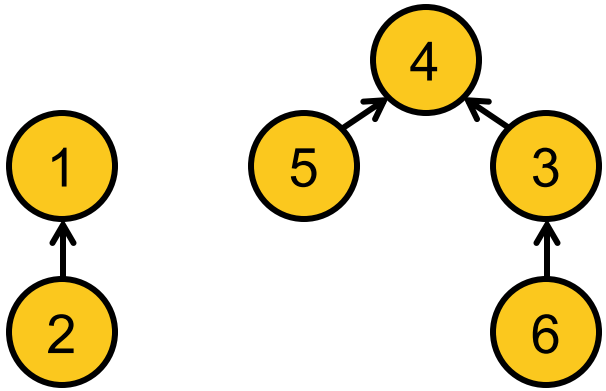
Operations:

- FIND(i): to which set does i belong?
- UNION(i,j): union the sets 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 = {}
- Sort edges such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$
- for each edge e_i={u,v} starting from i=1:
  - if FIND(u) ≠ FIND(v): # no cycle introduced?
    - F = F ∪ {{u,v}}
    - UNION(u,v)
- return F

Back to Runtime Considerations

- Sorting of edges needs O(|E| log |E|)
- forest: **Disjoint-set data structure**
  - initialization: O(|V|)
  - 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:

1. Algo always produces a spanning tree

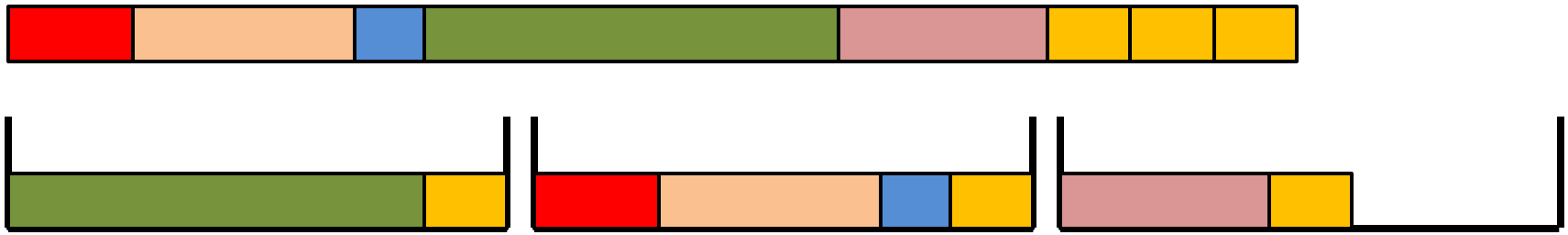
   final F contains no cycle and is connected by definition ✓

2. 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 a 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 F + e (i.e. P holds) ✓

Example 3: Bin Packing (BP)

Bin Packing Problem

Given a set of n items with sizes a_1, a_2, ..., a_n. Find an assignment of the a_i's to bins of size V such that the number of bins is minimal and the sum of the sizes of all items assigned to each bin is $\leq V$.



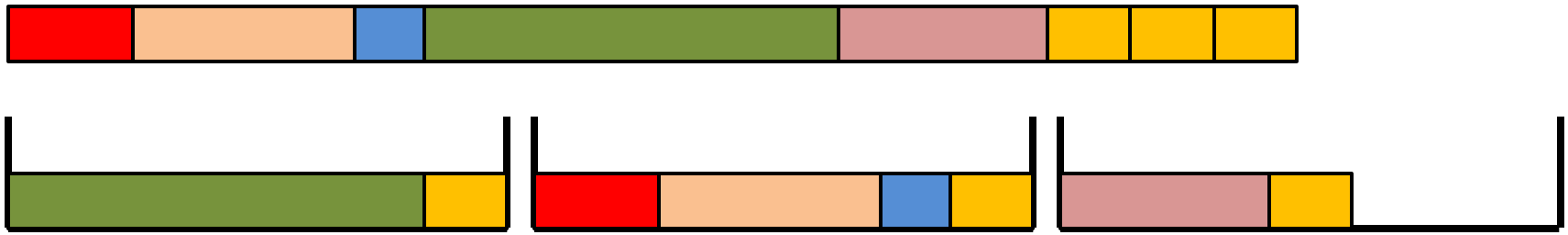
Applications

similar to multiprocessor scheduling of n jobs to m processors

Example 3: Bin Packing (BP)

Bin Packing Problem

Given a set of n items with sizes a_1, a_2, ..., a_n. Find an assignment of the a_i's to bins of size V such that the number of bins is minimal and the sum of the sizes of all items assigned to each bin is $\leq V$.



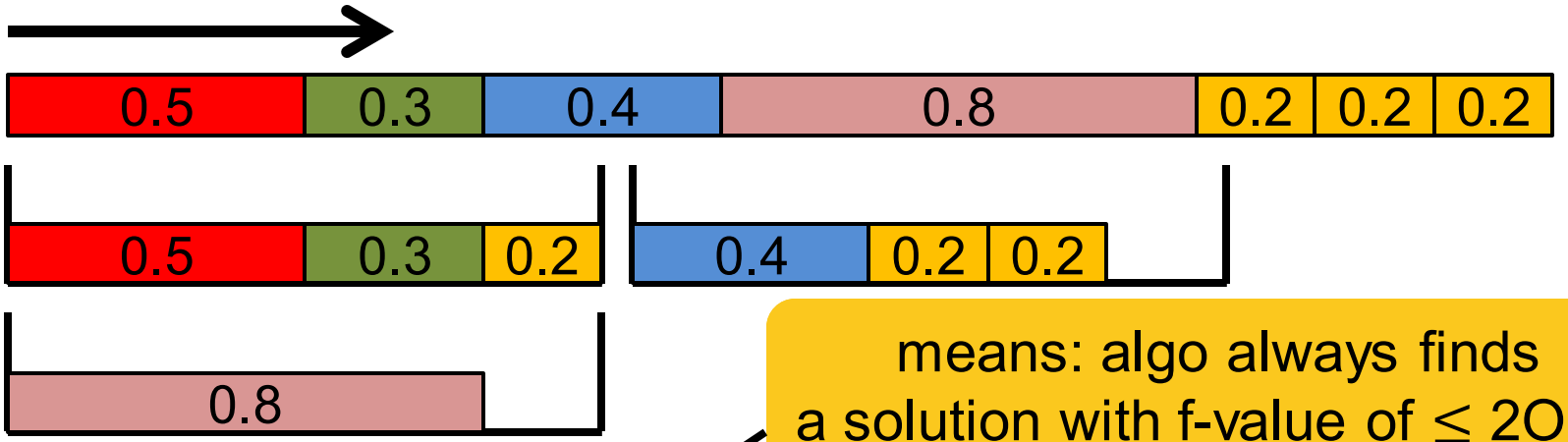
Known Facts

- no optimization algorithm reaches a better than 3/2 approximation in polynomial time (not shown here)
- greedy first-fit approach already yields an approximation algorithm with approximation ratio of 2

First-Fit Approach

First-Fit Algorithm

- without sorting the items do:
  - put each item into the first bin where it fits
  - if it does not fit anywhere, open a new bin



means: algo always finds a solution with f-value of $\leq$ 2OPT

Theorem: First-Fit algorithm is a 2-approximation algorithm

Proof: Assume First Fit uses m bins. Then, at least m-1 bins are more than half full (otherwise, move items).

$$\mathrm{OPT} > \frac{m-1}{2} \iff 2\mathrm{OPT} > m-1 \Longrightarrow 2\mathrm{OPT} \geq m$$

↑ because m and OPT are integer

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
  - see the example of bin packing
  - this is true in particular for so-called NP-hard problems

Obvious Question: when is greedy good?

Answer: if the problem is a matroid (not covered 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:

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 the Bellman equality and a table structure to store the optimal values (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)

Introduction to Optimization
Lecture 7: Discrete Optimization

October 26, 2018

TC2 - Optimisation

Université Paris-Saclay, Orsay, France

Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

1	Mon, 17.9.2018 Thu, 20.9.2018	Monday's lecture: introduction, example problems, problem types groups defined via wiki everybody went (actively!) through the Getting Started part of github.com/numbbo/coco
2	Fri, 21.9.2018	lecture "Benchmarking", final adjustments of groups everybody can run and postprocess the example experiment (~1h for final questions/help during the lecture)
3	Fri, 28.9.2018	lecture "Introduction to Continuous Optimization"
4	Fri, 5.10.2018	lecture "Gradient-Based Algorithms" + DFO
5	Fri, 12.10.2018	lecture "Stochastic Algorithms", in particular CMA-ES
6	Fri, 19.10.2018	lecture "Discrete Optimization I: graphs, greedy algos, dyn. progr." deadline for submitting data sets
	Wed, 24.10.2018	deadline for paper submission
7	Fri, 26.10.2018	final lecture "Discrete Optimization II: dyn. progr., B&B, heuristics"
	29.10.-2.11.2018	vacation aka learning for the exams
	Thu, 8.11.2018 / Fri, 9.11.2018	oral presentations (individual time slots)
	Fri, 16.11.2018	written exam

All deadlines:
23:59pm Paris time

Remaining Lecture Overview

- O-notation
- graphs
- greedy algorithms
- dynamic programming
- branch and bound
- randomized search heuristics

Reminder Dynamic Programming (DP)

Dynamic Programming

- exact algorithm
- solve problem via solutions of subproblems ("optimal substructure")
- not solving overlapping subproblems twice, but store solutions

Lecture Outline Dynamic Programming (DP)

What we will see:

❶ Example 1: The All-Pairs Shortest Path Problem

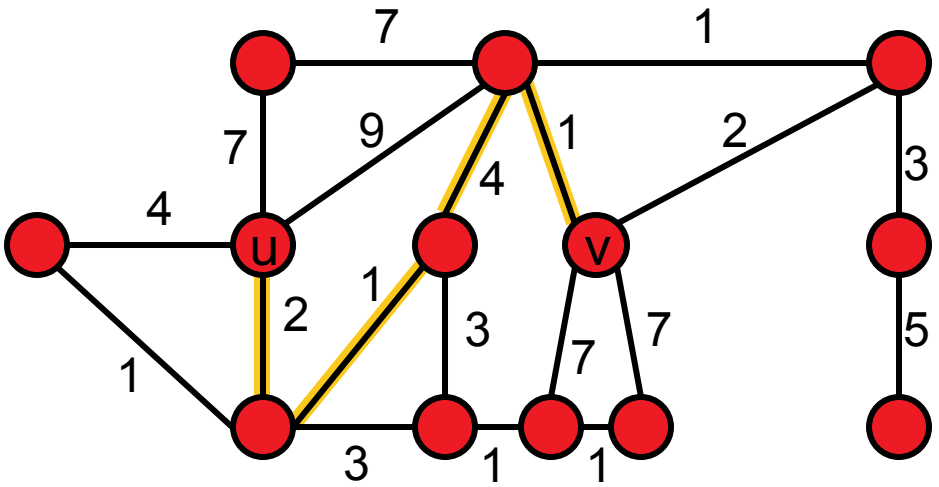
❷ Example 2: The knapsack problem

❸ Example 3: An approximation algorithm for 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

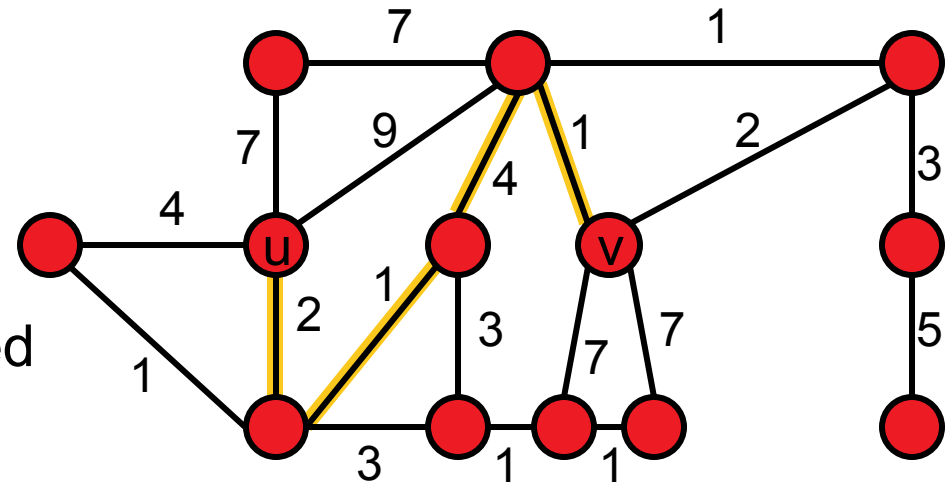
Finding routes for packages in a computer network

...

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:

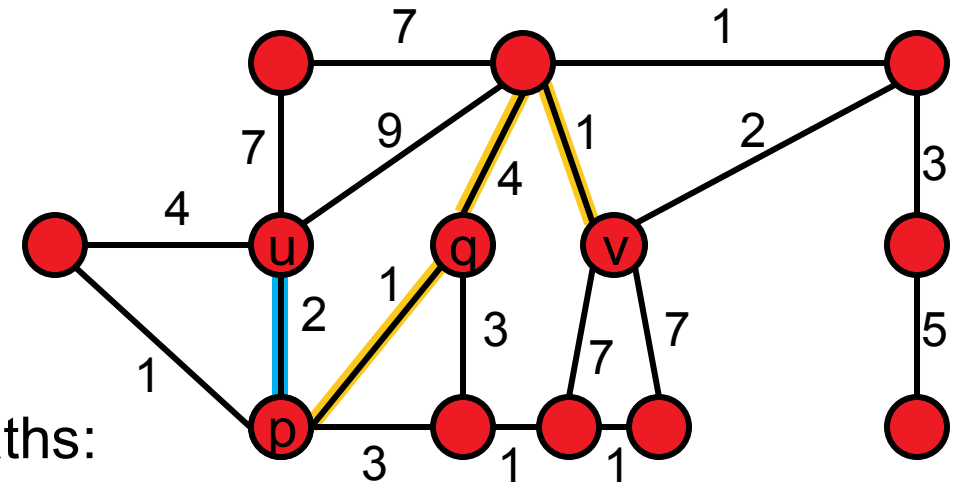
We can often assume that the edge weights are stored in a distance matrix D of dimension |E|x|E| where an entry $D_{i,j}$ gives the weight between nodes i and j and "non-edges" are assigned a value of ∞

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.



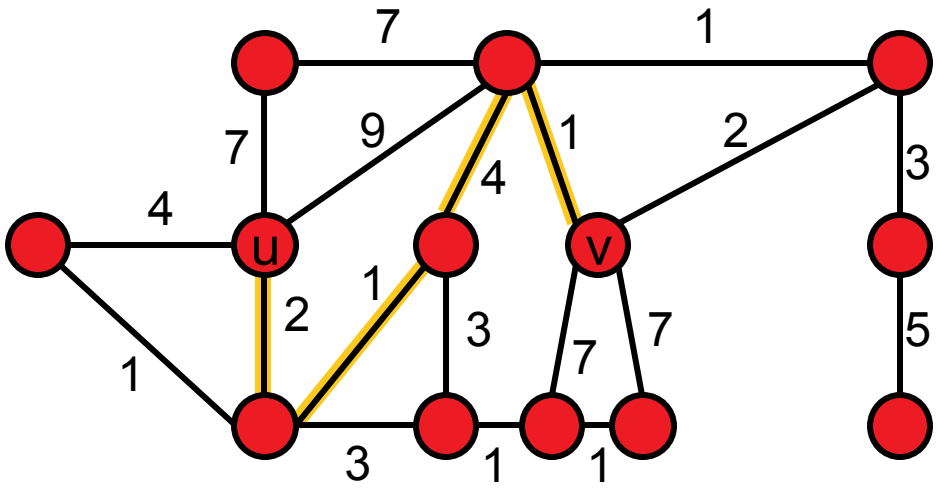
Overlapping Subproblems

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:

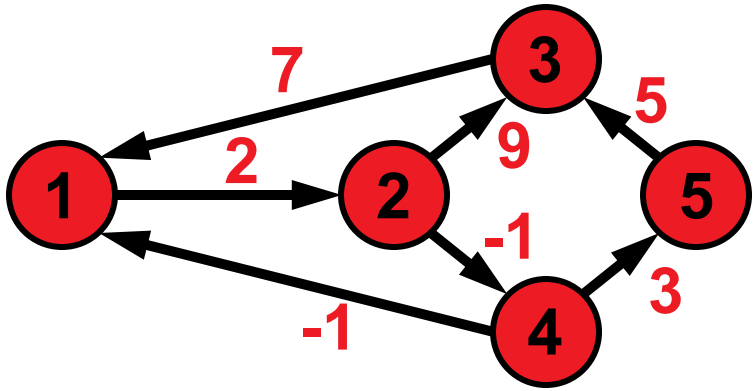
- if we knew that the shortest path between source and target goes through node v, we would be able to construct the optimal path from the shorter paths "source→v" and "v→target"
- subproblem P(k): compute all shortest paths where the intermediate nodes can be chosen from v_1, ..., v_k

AllPairsShortestPathFloyd(G, D)

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

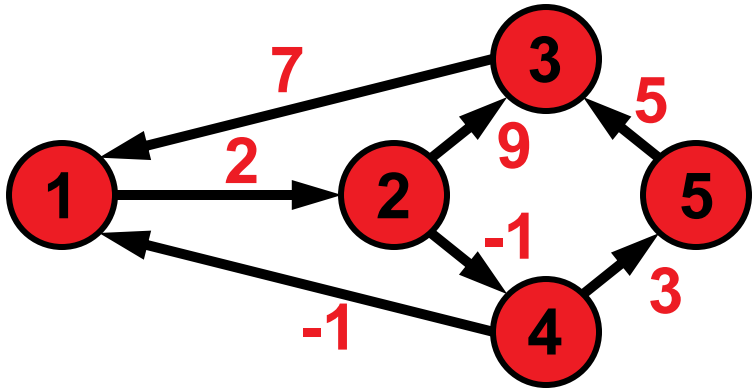
Note: Bernard Roy in 1959 and Stephen Warshall in 1962 essentially proposed the same algorithm independently.

Example



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

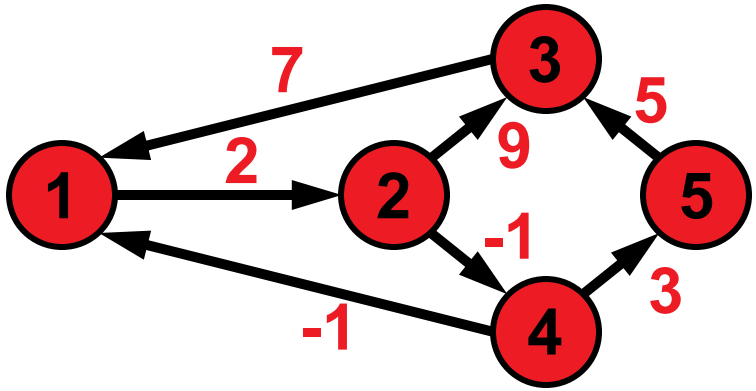
Example



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



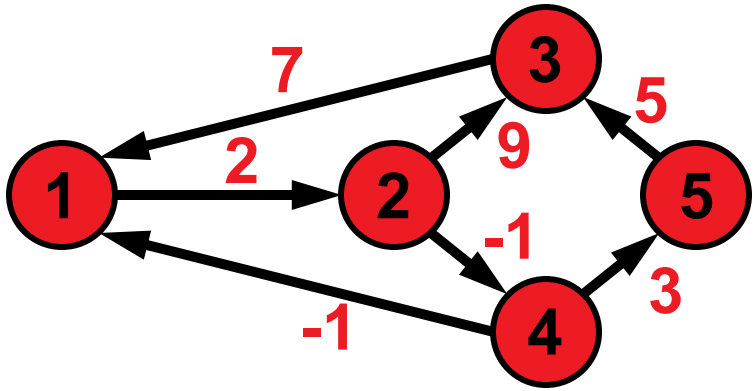
allow 1 as intermediate node

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



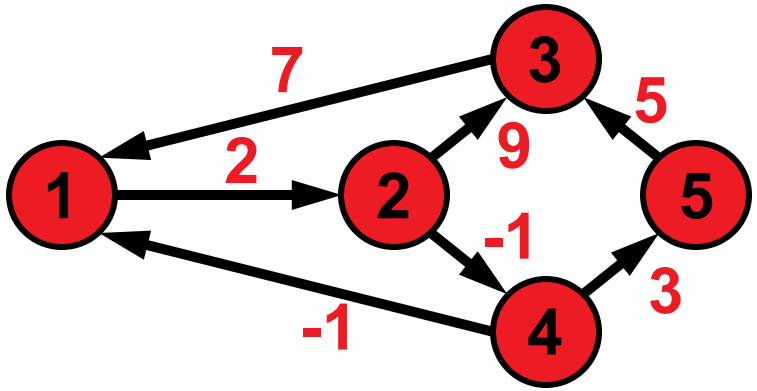
allow 1 as intermediate node

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



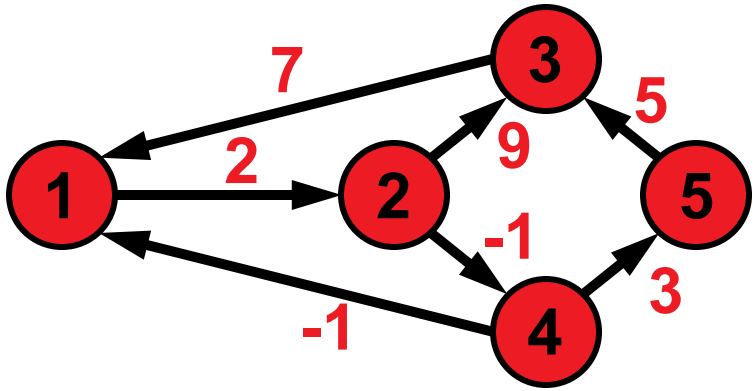
allow 1 as intermediate node

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



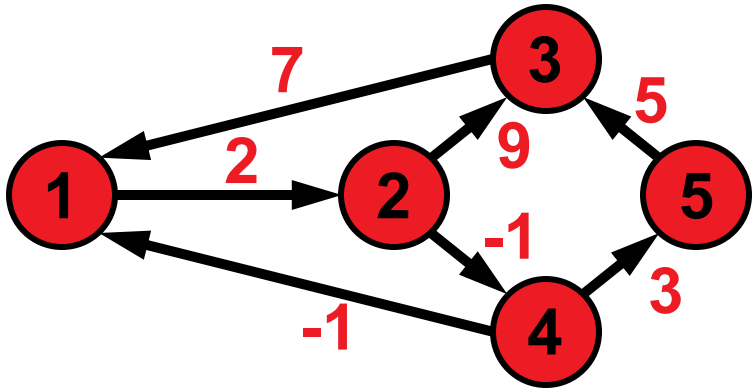
allow 1 as intermediate node

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		9			
4		1			
5					

Example

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



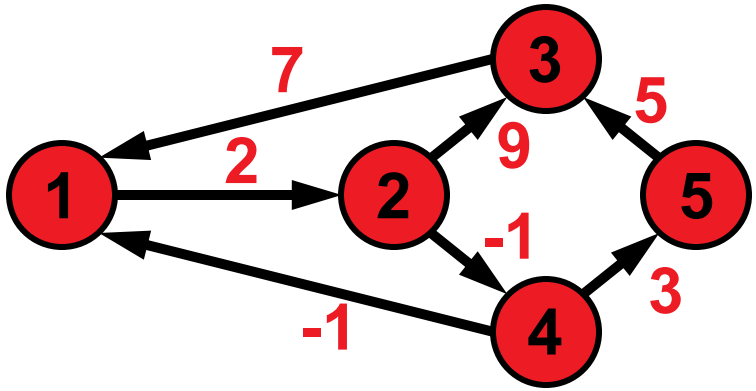
allow 1 as intermediate node

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



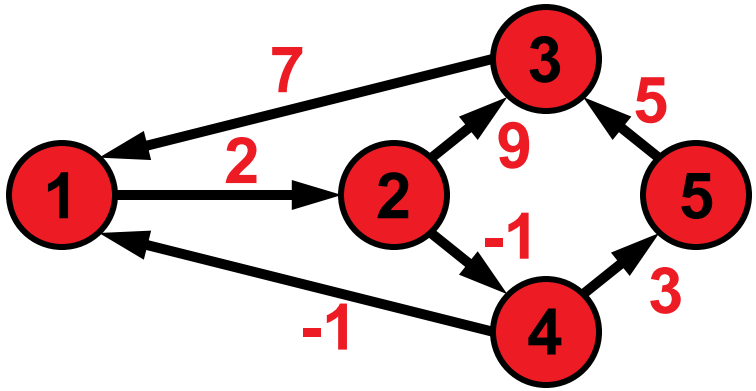
allow 1 & 2 as intermediate nodes

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



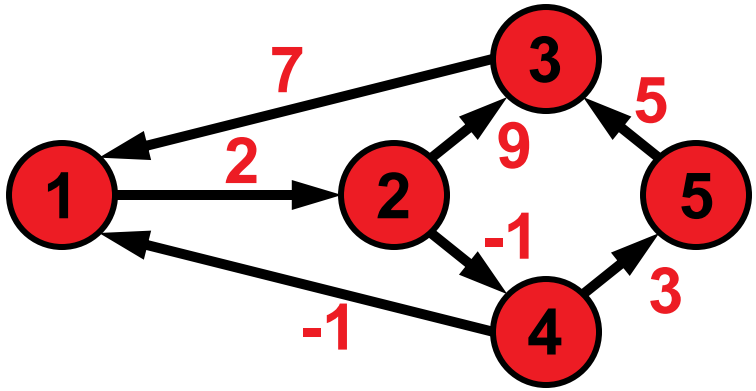
allow 1 & 2 as intermediate nodes

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



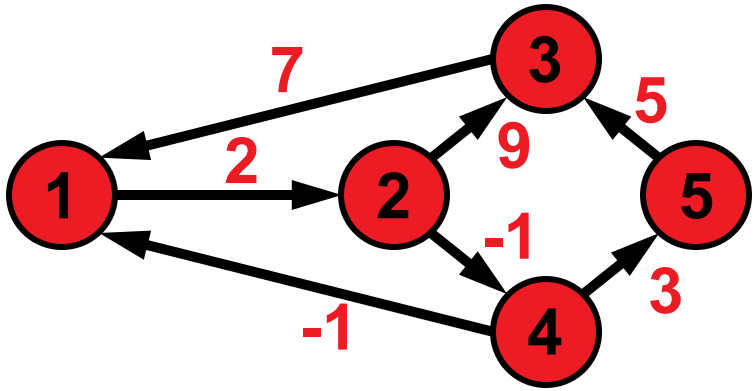
allow 1 & 2 as intermediate nodes

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



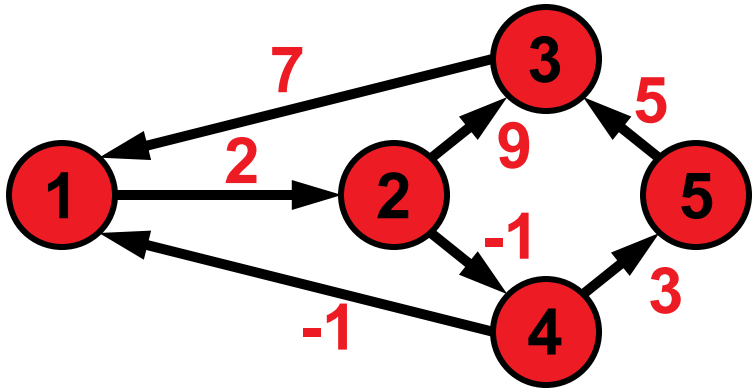
allow {1,2,3} as intermediate nodes

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



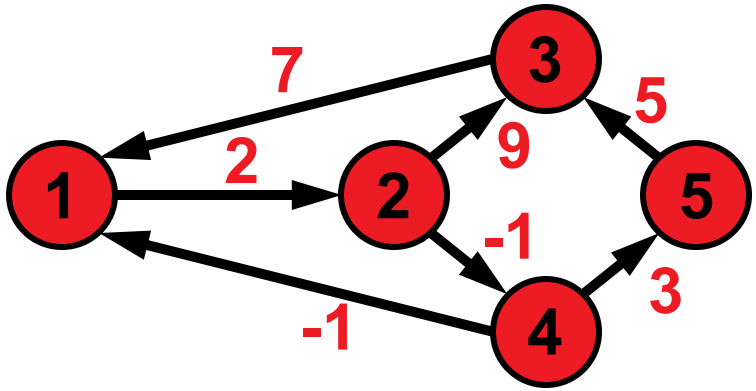
allow {1,2,3} as intermediate nodes

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



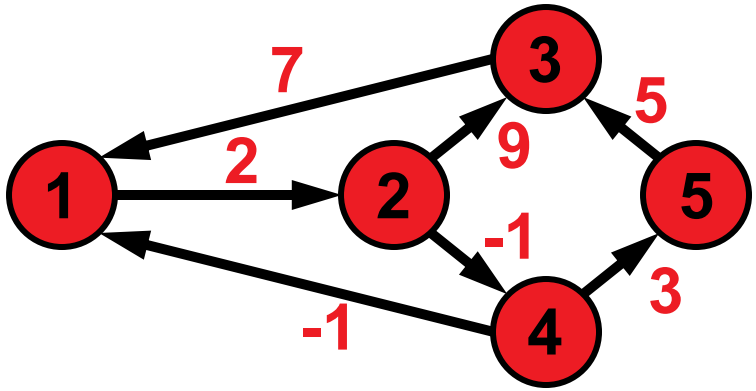
allow {1,2,3} as intermediate nodes

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



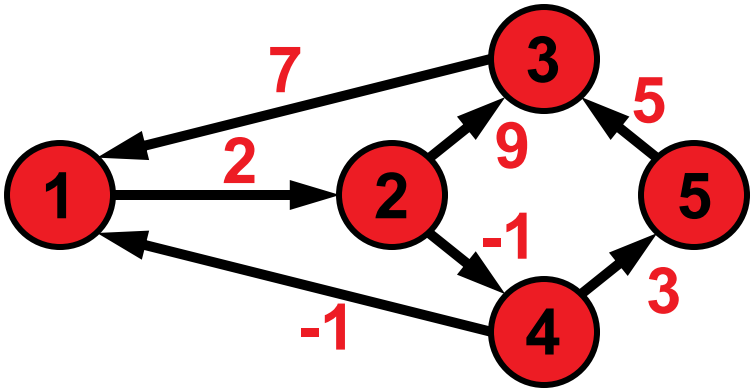
allow {1,2,3,4} as intermediate nodes

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



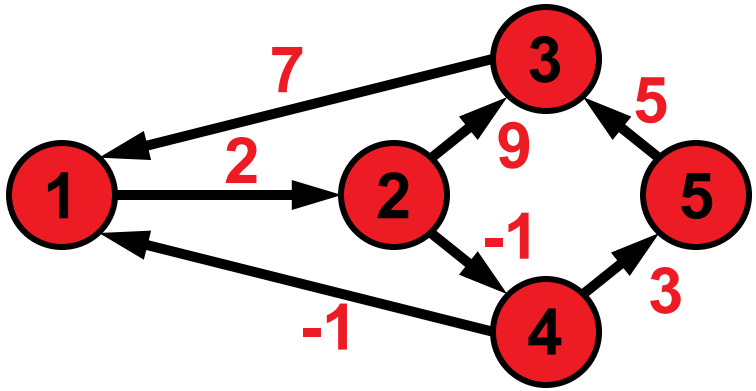
allow {1,2,3,4} as intermediate nodes

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



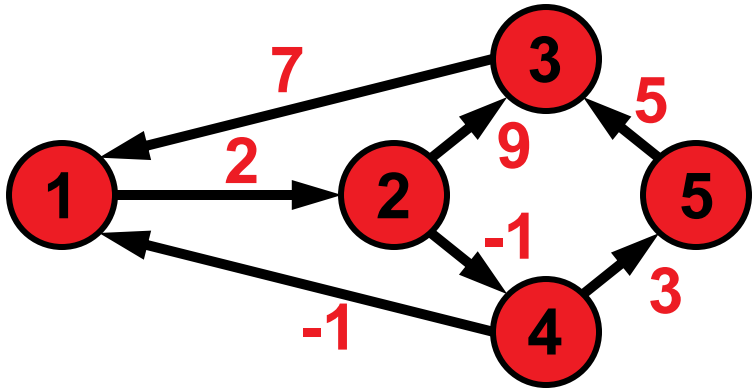
allow {1,2,3,4} as intermediate nodes

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



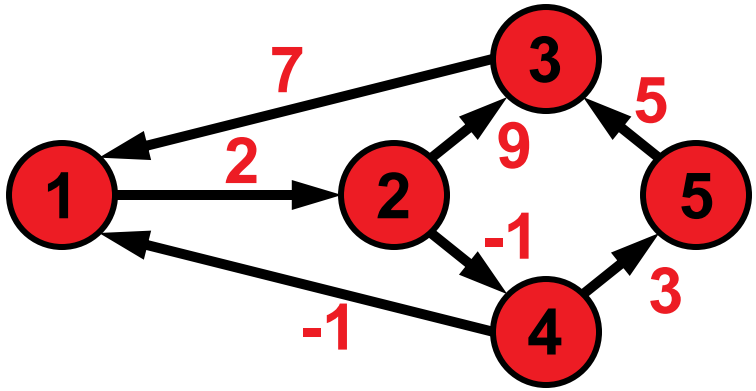
allow all nodes as intermediate nodes

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



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)

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

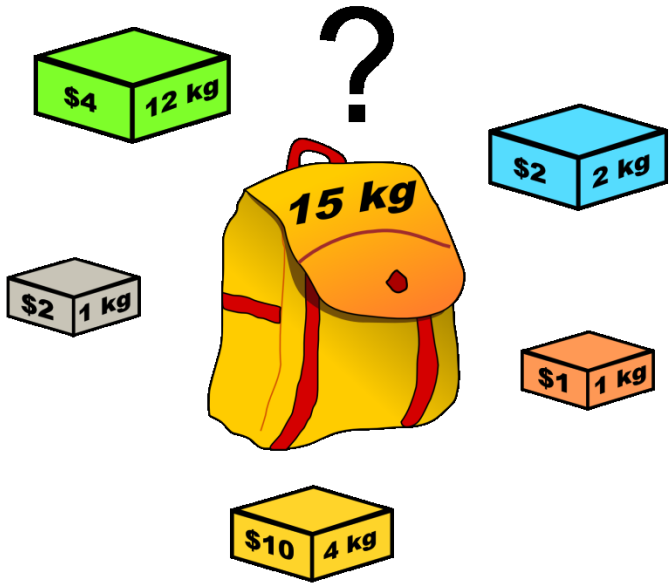
$$P_{i,j}(k) = \begin{cases} P_{i,j}(k-1) & \text{if } \mathrm{dist}(i,j) \leq \mathrm{dist}(i,k) + \mathrm{dist}(k,j) \\ P_{k,j}(k-1) & \text{if } \mathrm{dist}(i,j) > \mathrm{dist}(i,k) + \mathrm{dist}(k,j) \end{cases}$$

Example 2: The Knapsack Problem (KP)

Knapsack Problem

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

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



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 $1 \ldots i-1$:

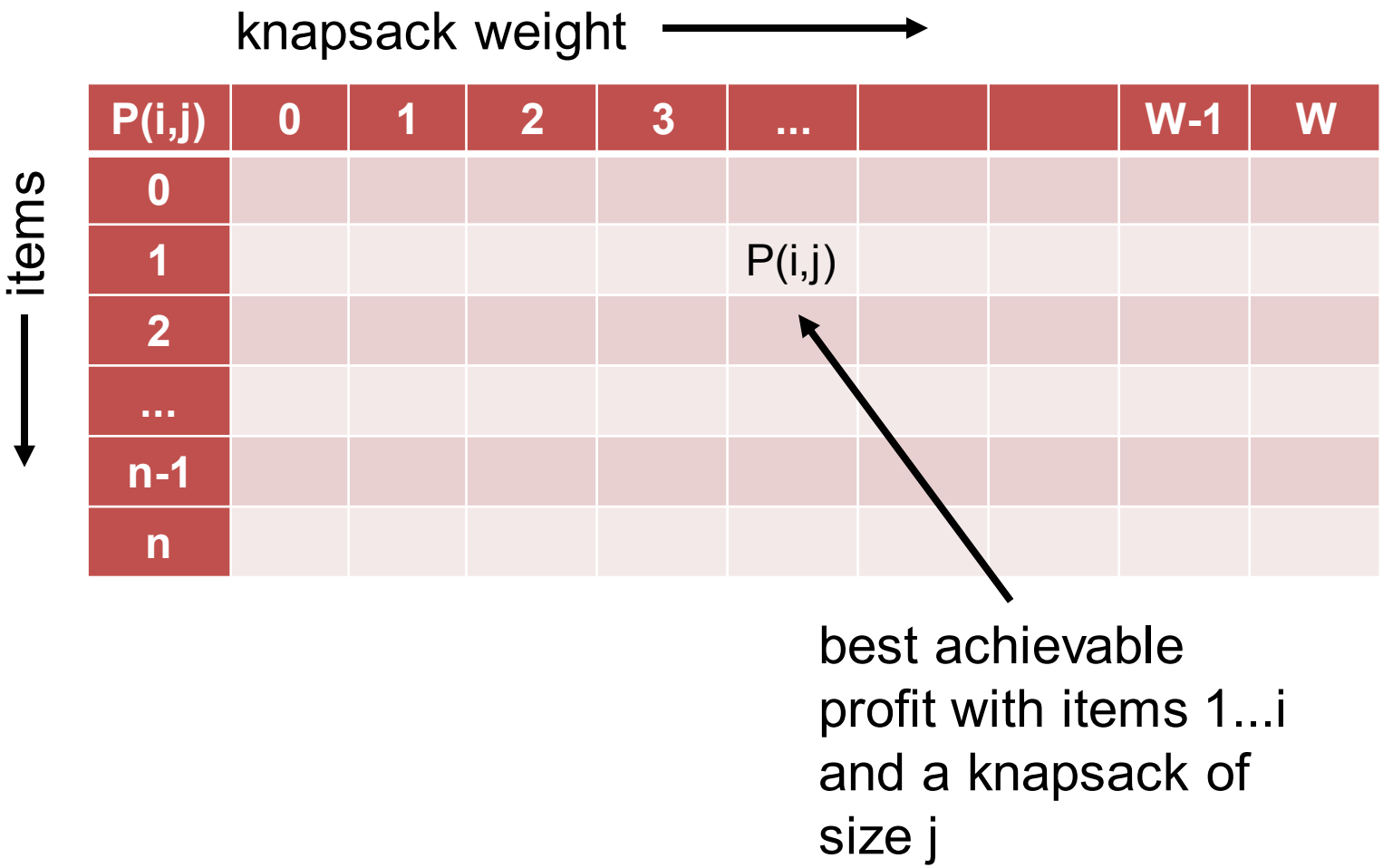
$$P(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

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)...



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

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:

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

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				
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 →

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	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					
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:

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

Example 3: Approximating the Knapsack Problem

- Simple to prove runtime of the previous algorithm as $O(nW)$
- In practice, i.e. for large n and W, computing the exact optimum for the knapsack problem might be too costly.

  no polynomial-time algo exists under famous P≠NP hypothesis
- If we want to have a polynomial-time algorithm, we have to trade for approximation quality.

Approximations, PTAS, and FPTAS

- An algorithm is a *ρ-approximation algorithm* for problem Π if, for each problem instance of Π, it outputs a feasible solution which function value is within a ratio ρ of the true optimum for that instance.

  remember the 2-approximation algo for bin packing?

- An algorithm A is an *approximation scheme* for a *maximization* problem Π if for any instance I of Π and a parameter ε>0, it outputs a solution s with $f_{\Pi}(I,s) \geq (1-\varepsilon) \cdot OPT$.

- An approximation scheme is called *polynomial time approximation scheme (PTAS)* if for a fixed ε>0, its running time is polynomially bounded in the size of the instance I.

  note: runtime might be exponential in 1/ε actually!

- An approximation scheme is a *fully polynomial time approximation scheme (FPTAS)* if its running time is bounded polynomially in both the size of the input I and in 1/ε.

An FPTAS for the Knapsack Problem

Similar to the previous dynamic programming algorithm, we can design a dynamic programming algorithm for which

- a subproblem is restricting the items to {1, ..., k} and searches for the lightest packing with prefixed profit P
- runs in $O(n^2 P_{\max})$ [idea: fill matrix of size $n \times nP_{\max}$]

What strange runtime is $O(n^2 P_{\max})$?

Answer: pseudo-polynomial (polynomial if $P_{\max}$ would be polynomial in input size)

Idea behind FPTAS:

- scale all profits p_i smartly to $\left\lfloor \frac{p_i n}{\varepsilon P_{\max}} \right\rfloor$ to make $P_{\max}$ polynomially bounded
- prove that dynamic programming approach computes profit of at least (1-ε)·OPT (not shown here)

Branch and Bound

Branch and Bound: General Ideas

- Systematic enumeration of candidate solutions in terms of a rooted tree
- Each tree node corresponds to a set of solutions; the whole search space on the root
- At each tree node, the corresponding subset of the search space is split into (non-overlapping) sub-subsets:
  - the optimum of the larger problem must be contained in at least one of its subproblems
- If tree nodes correspond to small enough subproblems, they are solved exhaustively.
- The smart part of the algorithm is the estimation of upper and lower bounds on the optimal function value achieved by solutions in the tree nodes
  - the exploration of a tree node is stopped ("pruning the tree") if a node's upper bound is already lower than the lower bound of an already explored node (assuming maximization)

Applying Branch and Bound

Needed for successful application of branch and bound:

- optimization problem
- finite (or at least countable) set of solutions
- clear idea of how to split problem into smaller subproblems
- efficient calculation of the following modules:
  - upper bound calculation
  - lower bound calculation

Computing Bounds (Maximization Problems)

Assume w.l.o.g. maximization of f(x) here

Lower Bounds

- any actual feasible solution will give a lower bound (which will be exact if the solution is the optimal one for the subproblem)
- hence, sampling a (feasible) solution can be one strategy
- using a heuristic to solve the subproblem another one

Upper Bounds

- upper bounds can be achieved by solving a relaxed version of the problem formulations (i.e. by either loosening or removing constraints)

Note: the better/tighter the bounds, the quicker the branch and bound tree can be pruned

Properties of Branch and Bound Algorithms

- Exact, global solver
- Can be slow; only exponential worst-case runtime
  - due to the exhaustive search behavior if no pruning of the search tree is possible
- but might work well in some cases

Advantages:

- can be stopped if lower and upper bound are “close enough” in practice (not necessarily exact anymore then)
- can be combined with other techniques, e.g. “branch and cut” (not covered here)

Example Branching Decisions

0-1 problems:

- choose unfixed variable x_i
- one subproblem defined by setting x_i to 0
- one subproblem defined by setting x_i to 1

General integer problem:

- choose unfixed variable x_i
- choose a value c that x_i can take
- one subproblem defined by restricting $x_i \leq c$
- one subproblem defined by restricting $x_i > c$

Combinatorial Problems:

- branching on certain discrete choices, e.g. an edge/vertex is chosen or not chosen

The branching decisions are then induced as additional constraints when defining the subproblems.

Which Tree Node to Branch on?

Several strategies (again assuming maximization):

- choose the subproblem with highest upper bound
  - gain the most in reducing overall upper bound
  - if upper bound not the optimal value, this problem needs to be branched upon anyway sooner or later
- choose the subproblem with lowest lower bound
- simple depth-first search or breadth-first search

see for example

https://en.wikipedia.org/wiki/Depth-first_search

https://en.wikipedia.org/wiki/Breadth-first_search

- problem-specific approach most likely to be a good choice

4 Steps Towards a Branch and Bound Algorithm

Concrete steps when designing a branch and bound algorithm:

- How to split a problem into subproblems (“branching”)?
- How to compute upper bounds (assuming maximization)?
- Optional: how to compute lower bounds?
- How to decide which next tree node to split?

Example: Application to ILPs

$$
\begin{aligned}
\text{maximize} \quad & c^T x \\
\text{subject to} \quad & Ax \leq b \\
& x \geq 0 \\
\text{and} \quad & x \in \mathbb{Z}^n
\end{aligned}
$$

The ILP formalization covers many problems such as

- Traveling Salesperson Person (TSP)
- Vertex Cover and other covering problems
- Set packing and other packing problems
- Boolean satisfiability (SAT)

Possible Ways to Solve an ILP

- Do not restrict the solutions to integers and round the solution found of the relaxed problem (=remove the integer constraints) by a continuous solver (i.e. solving the so-called *LP relaxation*)
  → no guarantee to be exact
- Exploiting the instance property of A being total unimodular:
  - feasible solutions are guaranteed to be integer in this case
  - algorithms for continuous relaxation can be used (e.g. the simplex algorithm)
- Using heuristic methods (typically without any quality guarantee)
  - we'll see these type of algorithms later in the lecture
- Using exact algorithms such as branch and bound

Branch and Bound for ILPs

Here, we just give an idea instead of a concrete algorithm...

- How to split a problem into subproblems (“branching”)?
- How to compute upper bounds (assuming maximization)?
- Optional: how to compute lower bounds?
- How to decide which next tree node to split?

Branch and Bound for ILPs

Here, we just give an idea instead of a concrete algorithm...

- How to compute upper bounds (assuming maximization)?
- How to split a problem into subproblems (“branching”)?
- Optional: how to compute lower bounds?
- How to decide which next tree node to split?

Branch and Bound for ILPs

How to compute upper bounds (assuming maximization)?

- drop the integer constraints and solve the so-called LP-relaxation
- can be done by standard LP algorithms such as `scipy.optimize.linprog` or Matlab's `linprog`

What's then?

- The LP has no feasible solution. Fine. Prune.
- We found an integer solution. Fine as well. Might give us a new lower bound to the overall problem.
- The LP problem has an optimal solution which is worse than the highest lower bound over all already explored subproblems. Fine. Prune.
- Otherwise: Branch on this subproblem: e.g. if optimal solution has $x_i=2.7865$, use $x_i \leq 2$ and $x_i \geq 3$ as new constraints

Branch and Bound for ILPs

How to split a problem into subproblems ("branching")?

- mainly needed if the solution of the LP-relaxation is not integer
- branch on a variable which is rational

Not discussed here in depth due to time:

- Optional: how to compute lower bounds?
- How to decide which next tree node to split?
  - seems to be good choice: subproblem with largest upper bound of LP-relaxation

Conclusions

I hope it became clear...

...what the algorithm design ideas of dynamic programming and branch and bound are

...and for which problem types they are supposed to be suitable

(Randomized) Search Heuristics

Motivation General Search Heuristics

- often, problem complicated and not much time available to develop a problem-specific algorithm
- search heuristics are a good choice:
  - relatively easy to implement
  - easy to adapt/change/improve
    - e.g. when the problem formulation changes in an early product design phase
    - or when slightly different problems need to be solved over time
- randomized/stochastic algorithms are a good choice because they are robust to noise

Lecture Outline Randomized Search Heuristics

Which algorithms will we touch?

1. Randomized Local Search (RLS)
2. Variable Neighborhood Search (VNS)
3. Tabu Search (TS)
4. Evolutionary Algorithms (EAs)

Neighborhoods

For most (stochastic) search heuristics, we need to define a *neighborhood structure*

- which search points are close to each other?

Example: k-bit flip / Hamming distance k neighborhood

- search space: bitstrings of length n ($\Omega=\{0,1\}^n$)
- two search points are neighbors if their Hamming distance is k
- in other words: x and y are neighbors if we can flip exactly k bits in x to obtain y
- 0001001101 is neighbor of
  0001000101 for k=1
  0101000101 for k=2
  1101000101 for k=3

Neighborhoods II

Example: possible neighborhoods for the knapsack problem

- search space again bitstrings of length n ($\Omega=\{0,1\}^n$)
- Hamming distance 1 neighborhood:
  - add an item or remove it from the packing
- replacing 2 items neighborhood:
  - replace one chosen item with an unchosen one
  - makes only sense in combination with other neighborhoods because the number of items stays constant
- Hamming distance 2 neighborhood on the contrary:
  - allows to change 2 arbitrary items, e.g.
    - add 2 new items
    - remove 2 chosen items
    - or replace one chosen item with an unchosen one

Randomized Local Search (RLS)

Idea behind (Randomized) Local Search:

- explore the local neighborhood of the current solution (randomly)

Pure Random Search:

- go to randomly chosen neighbor

First Improvement Local Search:

- go to first (randomly) chosen neighbor which is better

Best Improvement strategy:

- always go to the best neighbor
- not random anymore
- computationally expensive if neighborhood large

Variable Neighborhood Search

Main Idea: [Mladenovic and P. Hansen, 1997]

- change the neighborhood from time to time
  - local optima are not the same for different neighborhood operators
  - but often close to each other
  - global optimum is local optimum for all neighborhoods
- rather a framework than a concrete algorithm
  - e.g. deterministic and stochastic neighborhood changes
- typically combined with (i) first improvement, (ii) a random order in which the neighbors are visited and (iii) restarts

N. Mladenovic and P. Hansen (1997). "Variable neighborhood search". Computers and Operations Research 24 (11): 1097–1100.

Tabu Search

Disadvantages of local searches (with or without varying neighborhoods)

- they get stuck in local optima
- have problems to traverse large plateaus of equal objective function value ("random walk")

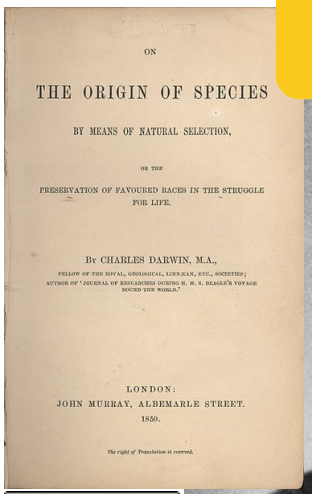
Tabu search addresses these by

- allowing worsening moves if all neighbors are explored
- introducing a tabu list of temporarily not allowed moves
- those restricted moves are
  - problem-specific and
  - can be specific solutions or not permitted "search directions" such as "don't include this edge anymore" or "do not flip this specific bit"
- the tabu list is typically restricted in size and after a while, restricted moves are permitted again

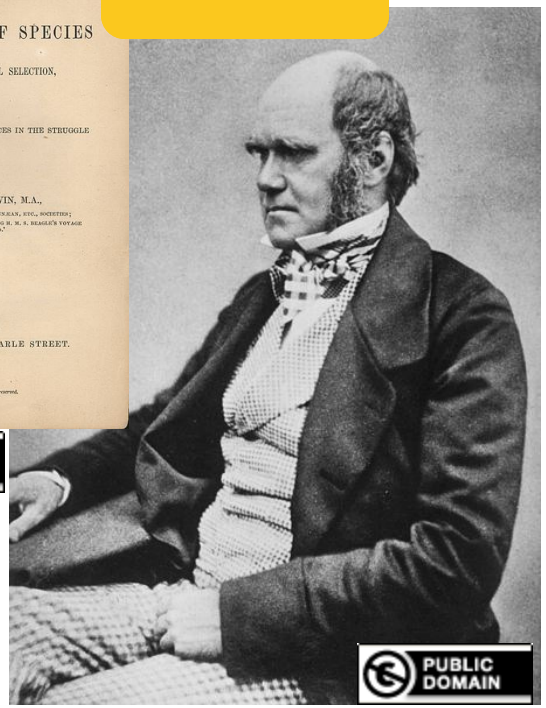
Stochastic Optimization Algorithms

One class of (bio-inspired) stochastic optimization algorithms: Evolutionary Algorithms (EAs)

- Class of optimization algorithms originally inspired by the idea of biological evolution
- selection, mutation, recombination



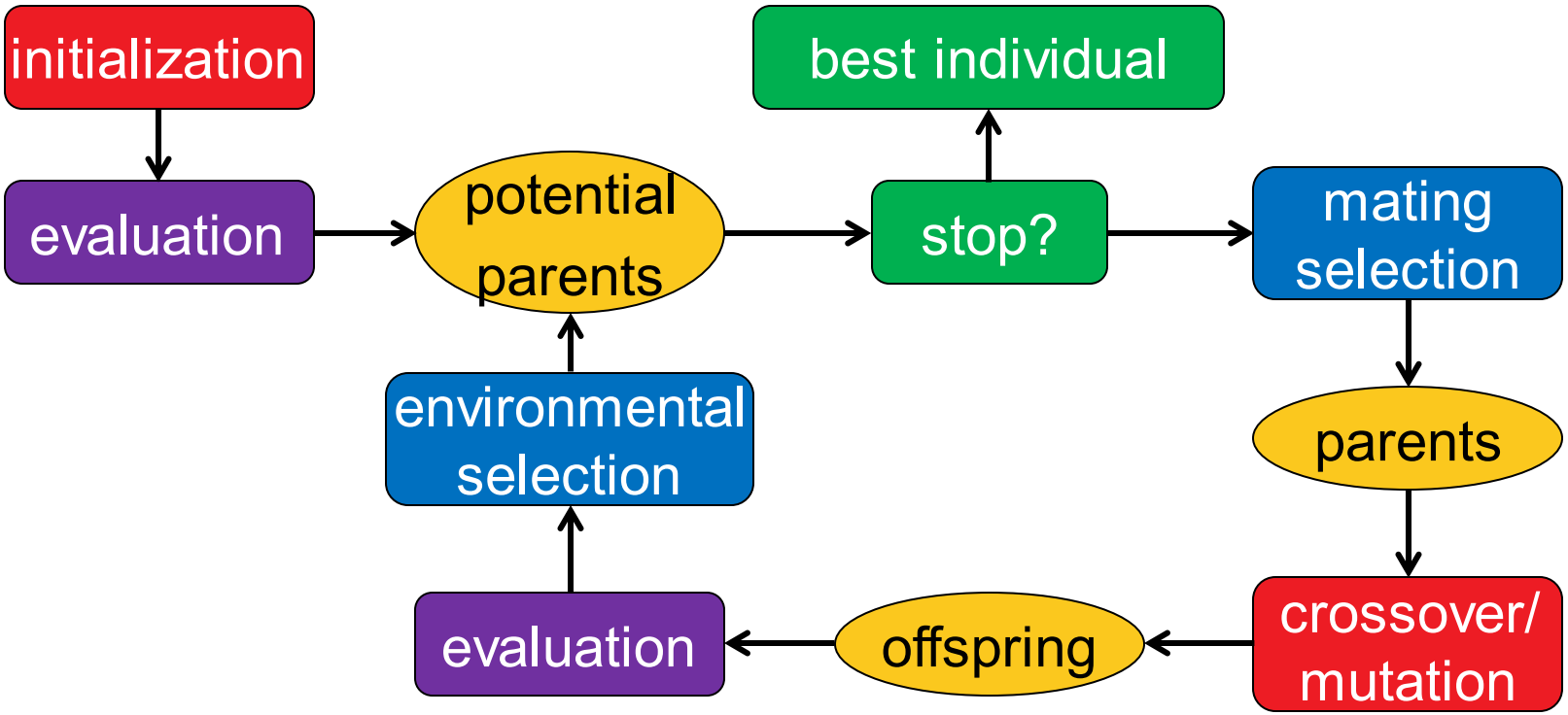
1859



Metaphors

Classical Optimization	Evolutionary Computation
variables or parameters	variables or chromosomes
candidate solution vector of decision variables / design variables / object variables	individual, offspring, parent
set of candidate solutions	population
objective function loss function cost function error function	fitness function
iteration	generation

Generic Framework of an EA



stochastic operators

"Darwinism"

stopping criteria

Important:
representation (search space)

The Historic Roots of EAs

Genetic Algorithms (GA)

J. Holland 1975 and D. Goldberg (USA)

$$\Omega = \{0,1\}^n$$

Evolution Strategies (ES)

I. Rechenberg and H.P. Schwefel, 1965 (Berlin)

$$\Omega = \mathbb{R}^n$$

Evolutionary Programming (EP)

L.J. Fogel 1966 (USA)

Genetic Programming (GP)

J. Koza 1990 (USA)

$$\Omega = \text{space of all programs}$$

nowadays one umbrella term: evolutionary algorithms

Genotype – Phenotype mapping

The genotype – phenotype mapping

- related to the question: how to come up with a fitness ("quality") of each individual from the representation?
- related to DNA vs. actual animal (which then has a fitness)

fitness of an individual not always = f(x)

- include constraints
- include diversity
- others
- but needed: always a total order on the solutions

Handling Constraints

Several possible ways to handle constraints, e.g.:

- resampling until a new feasible point is found ("often bad idea")
- penalty function approach: add constraint violation term (potentially scaled)
- repair approach: after generation of a new point, repair it (e.g. with a heuristic) to become feasible again if infeasible
  - continue to use repaired solution in the population or
  - use repaired solution only for the evaluation?
- multiobjective approach: keep objective function and constraint functions separate and try to optimize all of them in parallel
- ...

Examples for some EA parts

Selection

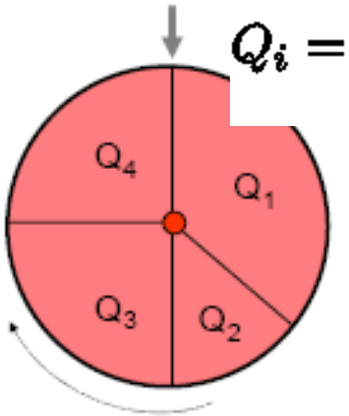
Selection is the major determinant for specifying the trade-off between exploitation and exploration

Selection is either

stochastic or deterministic

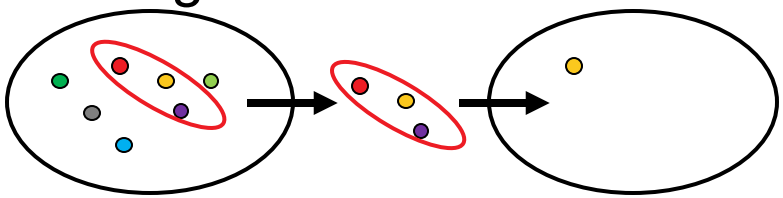
e.g. fitness proportional

$$Q_i = \frac{f(x_i)}{\sum_{j=1}^{\mu} f(x_j)}$$



Disadvantage: depends on scaling of f

e.g. via a tournament



e.g. (μ+λ), (μ,λ)

best μ from offspring *and* parents

best μ from offspring only

Mating selection (selection for variation): usually stochastic

Environmental selection (selection for survival): often deterministic

Variation Operators

Variation aims at generating new individuals on the basis of those individuals selected for mating

Variation = Mutation and Recombination/Crossover

mutation: *mut:* $\Omega \rightarrow \Omega$

recombination: *recomb:* $\Omega^r \rightarrow \Omega^s$ where $r \geq 2$ and $s \geq 1$

- choice always depends on the problem and the chosen representation
- however, there are some operators that are applicable to a wide range of problems and tailored to standard representations such as vectors, permutations, trees, etc.

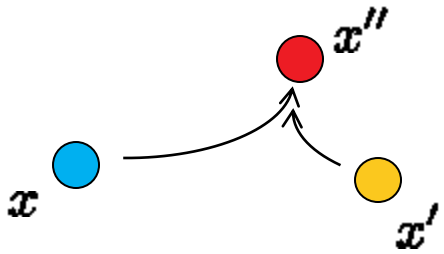
Variation Operators: Guidelines

Two desirable properties for mutation operators:

- every solution can be generation from every other with a probability greater than 0 (“exhaustiveness”)
- $d(x, x') < d(x, x'') => Prob(\text{mut}(x) = x') > Prob(\text{mut}(x) = x'')$ (“locality”)

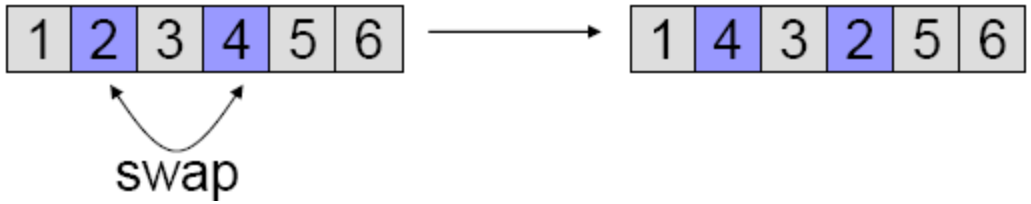
Desirable property of recombination operators (“in-between-ness”):

$$x'' = \text{recomb}(x, x') \Rightarrow d(x'', x) \leq d(x, x') \wedge d(x'', x') \leq d(x, x')$$

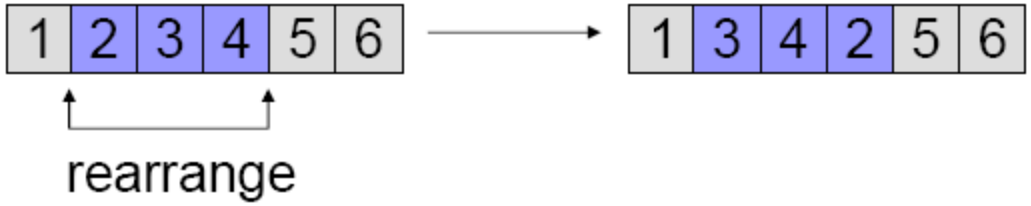


Examples of Mutation Operators on Permutations

Swap:



Scramble:

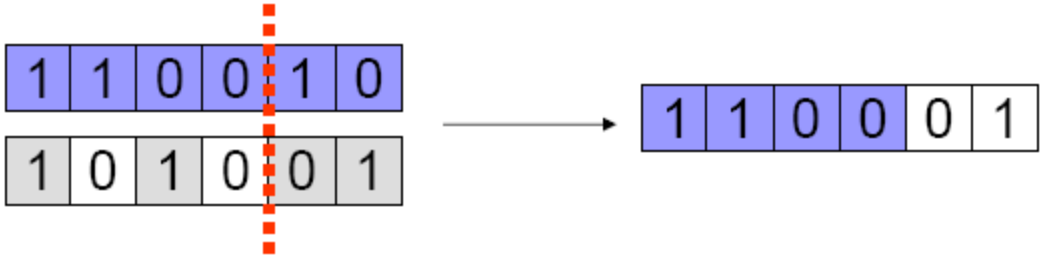


Invert:

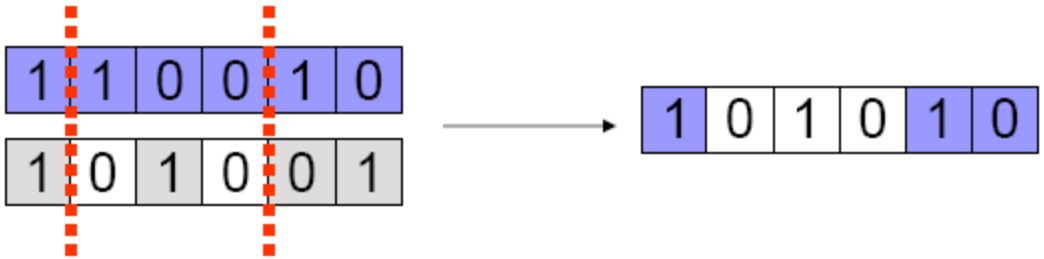
Insert:

Examples of Recombination Operators: $\{0,1\}^n$

1-point crossover



n-point crossover



uniform crossover

choose each bit independently from one parent or another

A Canonical Genetic Algorithm

- binary search space, maximization
- uniform initialization
- generational cycle: of the population
  - evaluation of solutions
  - mating selection (e.g. roulette wheel)
  - crossover (e.g. 1-point)
  - environmental selection (e.g. plus-selection)

Conclusions

- EAs are generic algorithms (randomized search heuristics, meta-heuristics, ...) for black box optimization

  no or almost no assumptions on the objective function

- They are typically less efficient than problem-specific (exact) algorithms (in terms of #funevals)

  less differences in the continuous case (as we have seen)

- Allow for an easy and rapid implementation and therefore to find good solutions fast

  easy to incorporate (and recommended!) to incorporate problem-specific knowledge to improve the algorithm

Conclusions

I hope it became clear...

...that heuristics is what we typically can afford in practice (no guarantees and no proofs)

...what are the main ideas behind evolutionary algorithms

...and that evolutionary algorithms and genetic algorithms are no synonyms