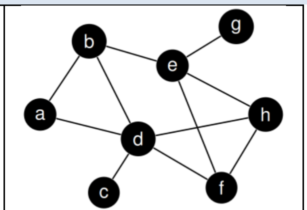
COMBINATORIAL PROBLEMS - P-CLASS

Graph Search

Given: $G = (V, E)$, start node
Goal: Search in a graph



DFS Depth-First-Search

1. Start at a, put it on stack.
Stack = LIFO "Last In - First Out"
2. Whenever there is an unmarked neighbour, go there and and put it on stack
3. If there is no unmarked neighbour, backtrack; i.e. remove current node from stack (grey ⇒ green) and go to step 2.

Insert from top ↓ Access from top ↓															
								g		h					
							e	e	e	e	e				
				c		f	f	f	f	f	f	f			
			d	d	d	d	d	d	d	d	d	d	d		
		b	b	b	b	b	b	b	b	b	b	b	b	b	
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

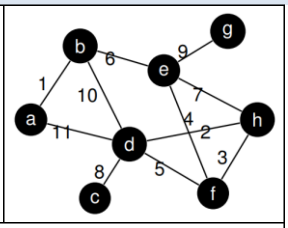
BFS Breadth-First Search

1. Start at a, put it in queue.
Queue = FIFO "First In - First Out"
2. Output first vertex from queue (grey ⇒ green). Mark all neighbors and put them in queue (white ⇒ grey). Do so until queue is empty

Insert from top ↓ / Access from bottom ↑								
				h	g			
				f	h	g		
		d	e	c	f	h	g	
	a	b	d	e	c	f	h	g

Minimum Spanning Tree (MST)

Given: Graph $G = (V, E, W)$ with undirected edges set E, with positive weights W
Goal: Find a set of edges that connects all vertices of G and has minimum total weight.
Application: Network design (water pipes, electricity cables, chip design)
Algorithm: Kruskal's, Prim's, Optimistic, Pessimistic



Optimistic Approach =Kruskal's algorithm 1956 (Greedy)

$O(|E| + \log|E|)$ with union-find-datastructure

Successively build the cheapest connection available that is not redundant.

```
Sort edges of G by increasing weight
Set E_T to ∅
For k = 1..n_e, do:
   If E_T ∪ {e_k} has no cycle
      Set E_T = E_T ∪ {e_k}
Return T = (V, E_T)
```

ab	dh	fh	ef	df	be	eh	cd	eg	bd	ad	total
1	2	3	4	5	6	7	8	9	10	11	33

Pessimistic Approach

Successively rule out the most expensive line that is not absolutely needed.

ad	bd	eg	cd	eh	be	df	ef	fh	dh	ab	total
11	10	9	8	7	6	5	4	3	2	1	33

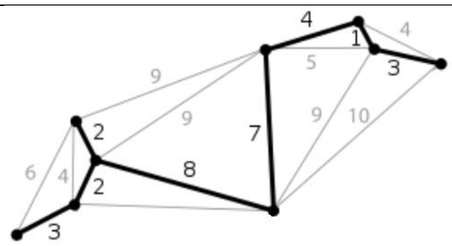
Prim's Algorithm 1957 (Greedy)

$O(|E| + |V| \log|V|)$ if L is managed with a Brodal queue

Choose an arbitrary start vertex v_0 and set $M = \{v_0\}$. Iteratively add to M a vertex in $V \setminus M$ that can be reached the cheapest from the current set M. Select the corresponding edge. Continue until $M = V$.

a	b	e	f	h	d	c	g	total
	a-b	b-e	e-f	f-h	h-d	d-c	e-g	
	1	6	4	3	2	8	9	33

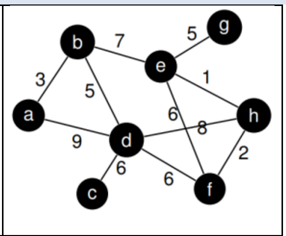
```
For each vertex u ∈ V do:
   λ[u] = ∞
   p[u] = emptyset
   Choose a vertex s and set λ[s] ≔ 0
   E_T = emptyset
   L = V    // List of vertices not yet in T
While L ≠ ∅ do
   Remove from L the vertex u with lower λ[u]
   If u ≠ s then
      E_T ≔ E_T ∪ {p[u], u}
   For each vertex v ∈ adj[u] do
      If v ∈ L and λ[v] > c_uv then
         λ[v] = c_uv
         p[v] = u
Return T = (V, E_T)
```



Shortest paths

Given: Graph $G = (V, E, C)$ with cost $c_{ij} \geq 0$ for each edge $e \in E$ and a start vertex $s \in V$
Goal 1: Find the shortest path from start s to j.
Goal 2: Find the shortest path from start s to all other vertices.
Goal 3: Find the shortest path between all pairs of vertices.

Algorithm: (Goal 1+2) Dijkstra's, (Goal 3) Floyd-Warshall



Dijkstra's algorithm 1959

$G = (V, E, C)$

$O(|E| + |V| \log|V|)$ if L is managed with a Brodal queue

We iteratively compute the shortest distance $I(v)$ for the vertex v closest to v_0 that has not been reached yet.
Not working with negative weights.

```
λj: length of the shortest path
pj: predecessor of j on the shortest path
s: start vertex
For all j ∈ V do λj = ∞; pj = ∅
λs = 0; L = V
While L ≠ ∅
   Find i such that λi = min(λk | k ∈ L)
   L = L ohne {i}
   For all j ∈ succ[i] do
      If j ∈ L and λj > λi + cij
         λj = λi + cij;
         pj = i
Return Lenghts λ and predecessors p
```

a	b=3	d=8	c=14	c=14	c=14	c=14	g=15
	d=9	e=10	e=10	f=14	f=13	g=15	
			f=14	h=11	g=15		
			h=16	g=15			

Floyd-Warshall algorithm

relies on dynamic programming

use recursion

negative weighs allowed as long as no negative cycles

$minPath(i, j, n)$ are the shortest distances from v_i to v_j, ∞ if not existing.
Iterate through all rows/columns:

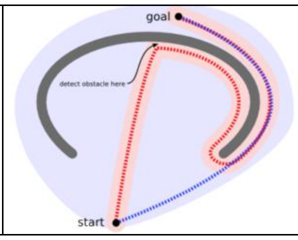
$$minPath(i, j, k+1) = \min \begin{cases} minPath(i, j, k) \\ minPath(i, k+1, k) + \\ minPath(k+1, j, k) \end{cases}$$

$i = 1$

	a	b	c	d	e	f	g	h
a	–	3	∞	9 → 8	∞	∞	∞	∞
b	3	–	∞	5	7	∞	∞	∞
c	∞	∞	–	6	∞	∞	∞	∞
d	9 → 8	5	6	–	∞ → 12	6	∞	8
e	∞	7	∞	∞ → 12	–	6	5	1
f	∞	∞	∞	6	6	–	∞	2
g	∞	∞	∞	∞	5	∞	–	∞
h	∞	∞	∞	8	1	2	∞	–

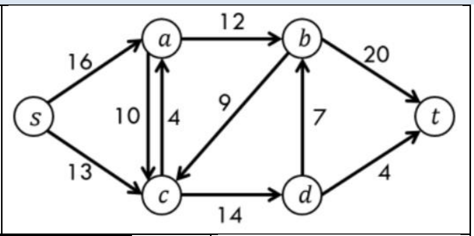
Var: Pathfinding

Given: start and goal coordinates
Problem: We only see the immediate neighborhood of our position.

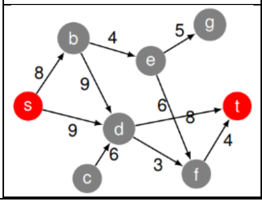
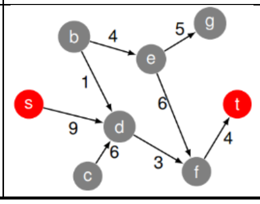
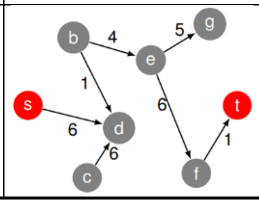


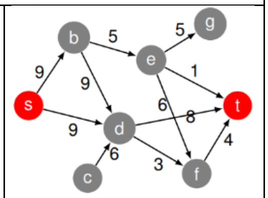
Max-Flow

Input: directed graph $G = (V, E)$ with capacity $c(e) > 0$
Two special nodes, source s, and ,sink/target t, are given, $s \neq t$
Goal: Maximize the total amount of flow from s to t,
where the flow on edge e doesn't exceed $c(e)$ and for every node $v \neq s \text{ and } v \neq t$, incoming flow is equal to outgoing flow.
Applications: Network routing, Transportation, Bipartite Match
Algorithm: Ford-Fulkerson-Alg, Edmonds-Karp



Greedily finding augmenting paths

s-b-e-t	s-b-d-t	s-d-f-t	total
1	8	3	12
s-b: $9 \rightarrow 8$ b-e: $5 \rightarrow 4$ e-t: $1 \rightarrow 0$	s-b: $8 \rightarrow 0$ b-d: $9 \rightarrow 1$ d-t: $8 \rightarrow 0$	s-d: $9 \rightarrow 6$ d-f: $3 \rightarrow 0$ f-t: $4 \rightarrow 1$	
			

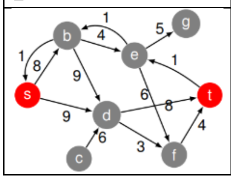
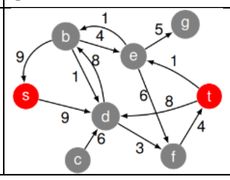
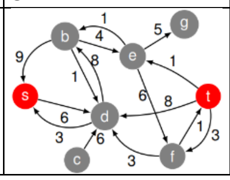
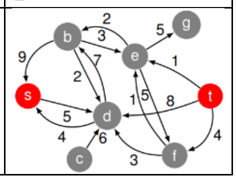


Ford-Fulkerson

$O((n_v + n_e)\, f^*)$
f^*: optimal flow

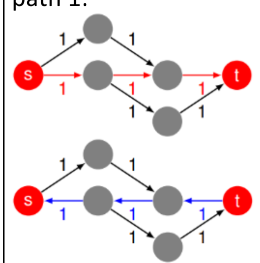
Idea: Insert backwards edges that can be used to (partially) undo previous paths.

```
Set f_total = 0
While there is a path from s to t in G:
   Determine smallest capacity g along the path P
   Add f to f_total
   Foreach edge u→v on the path
      decrease c(u→v) by f
      increase c(v→u) by f (backward)
      delete edge if capacity c = 0
```

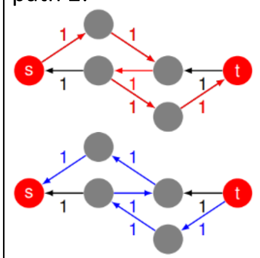
s-b-e-t	s-b-d-t	s-d-f-t	s-d-b-e-f-t
1	8	3	1
			

Problem: Inefficient behaviour if the augmenting path is chosen arbitrarily.
Solution: Edmonds-Karp algorithm

path 1:



path 2:



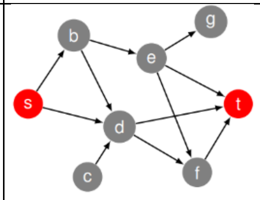
Edmonds-Karp algorithm

starting at s

Idea: In each iteration, use a shortest augmenting path, i.e. a path from s to t with the fewest number of edges. Can be found with BFS in time $O(|E|)$.

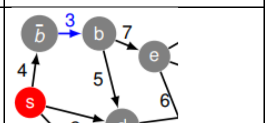
Insert from top ↓				t=2
		d=1	e=2	f=2
Access from bottom ↑	s=0	b=1	d=1	e=2

shortest augmenting path: s-d-t



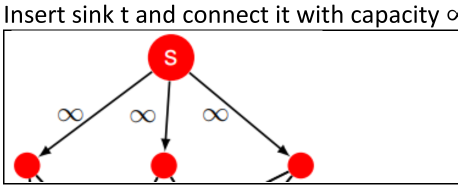
with vertex restrictions

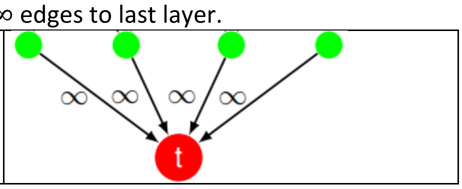
Restriction: Maximum flow through vertices is restricted.
Solution: Add additional vertex before the restricted vertex and add new edge with weight = vertex restriction. Direct all ingoing edges to the new vertex. e.g. b by 3:

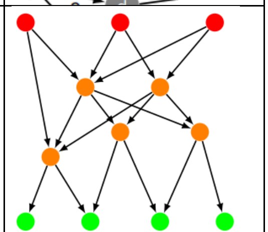


Distribution problem reduced to Max-Flow

Insert source s and connect it with capacity ∞ edges to first layer.
Insert sink t and connect it with capacity ∞ edges to last layer.

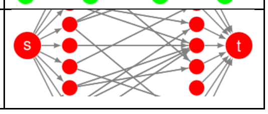


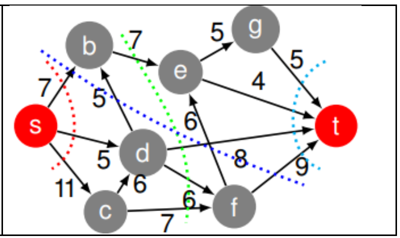


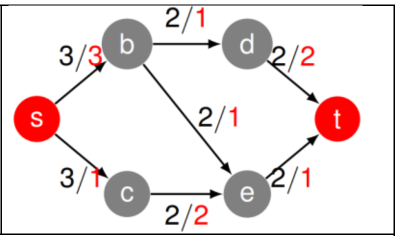


Bipartite matching

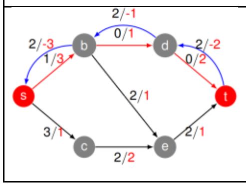
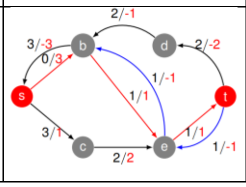
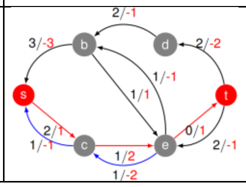
Make edges directed. Add source and sink vertices s and t with corresponding edges. Set all edge weights to 1. Find integer maximum flow with Ford-Fulkerson algorithm.



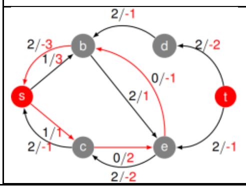
Var: **Max-Flow** **Min-Cut**	**Goal**: Find a minimum partition of the vertex set which divides S and T into two sets. **Applicationgs**: Redundancy in networks, reliability in telecommunication, military strategy, optimizing traffic systems The maximum st flow equals the minimum value of all (exponentially many) possible st-cuts.	

Var: **Min-Cost** **Max-Flow** harder	Each edge e has capacity $c(e)$ and cost $cost(e)$ **Goal**: Find the maximum flow that has the minimum total cost	

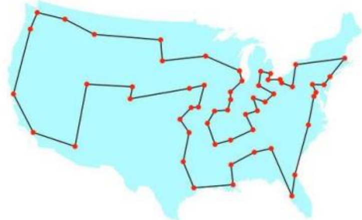
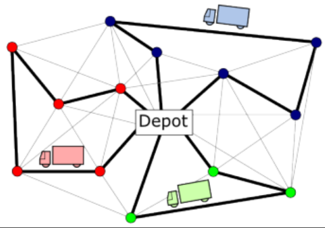
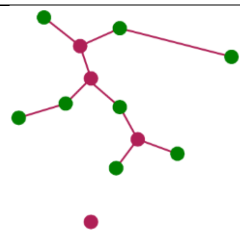
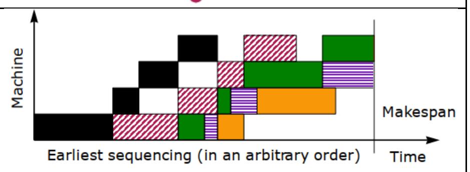
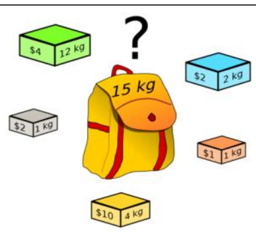
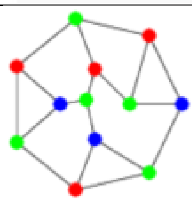
Often there are several possible maximum flows with a cost per flow unit.
Solution: Extend Ford-Fulkerson algorithm to take cost criterion into account.

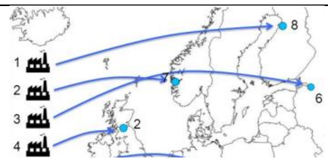
s-b-d-t	s-b-e-t	s-c-e-t	
2*6=16	1*5=5	1*4	
			capacity = 4 cost = 21

Until now, we just used the Ford-Fulkerson algorithm with additionally annotating the costs. Now, look for cycles, respecting the capacities. This does not change the total flow from s to t, but decreases the total cost.

s-c-e-b-s	
capacity 1*(3-4)	
s-b-e → s-c-e	
	capacity = 4 new cost = 20

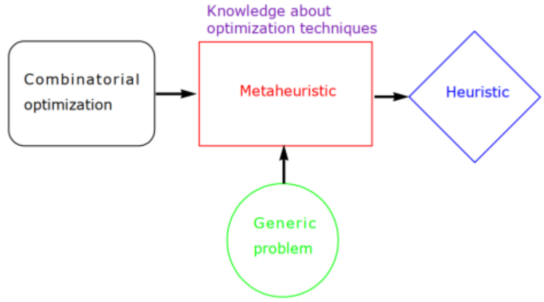
COMBINATORIAL PROBLEMS - NP-CLASS

TSP **Traveling Salesperson Problem** $O\left((n-1)!\right)$	**Input**: n cities, d_{ij} distances matrix between cities i and j. **Goal**: Find the shortes tour passing exactly once in each city. $$\sum_{i=1}^{n-1} d_{p_i p_{i+1}} + d_{p_n p_1}$$ **Application**: Network Design, pipes, cables, chip design, ... **Algorithms**: Nearest Neighbour (=Greedy), MST-based $MST < TSP \leq 2 * MST$	
CVRP **Capacitated Vehicle Routing Problem**	**Input:** n customers and 1 depot, q_i: quantity ordered by customer i, distances d_{ij}, Q verhicle capacity **Goal**: Minimize the total length performed by the vehicle Each customer gets his delivery, Each tour start and ends at the depot Total demand on each tour des not exceed Q	
Var:	**Goal:** Minimize number of trucks/paths	
Steiner Tree Problem	**Input**: $G = (V, E, C)$ with vertices, edges and weights and $subset\ D \subseteq V$ **Goal**: Find a tree of minimum weight containing all vertices of D and, eventually, other vertices of V not in D (Steiner nodes).	
Scheduling Problems	**Given:** n jobs and m machines, each job contains m tasks Machine has to be free and previous task done. Each task has a known processing time p_{ij} on the machine. **Goal**: Find a "good" order in which the jobs should be processed. **Objectives**: - Makespan: Minimize the completion time of the last job - Sum of completion times: (weighted) sum of completion times of all jobs - Minimum Tardiness: minimize the individual due date.	
Var: Permutation Flow Shop Problem	Huge objects or production line Ordering of tasks on machines is fixed. n! permutations	
Var: Flow Shop Problem	Smaller objects which can be storaged	
Var: Job Shop Problem	Introduction for new employies	
Knapsack Packing Problem	**Input:** n items each with a weight w_i and a value v_i, maximum weight capacity W **Goal**: maximize the value with respect to maximum weight **Algorithms**: Naive Greedy, Smarter Greedy simplest ILP (only one inequality) but NP-complete	
Santa Claus Problem	**Input:** n item with location on earth and weigt, max carry weight **Goal:** Minimize the weighted distance to deliver all items to the given location	
Vertex Coloring	**Input**: Graph **Goal**: Find an assignment of colors to each vertex such that no edge connects two identically colored vertices. **Variants**: Minimize the number of colors **Algorithm**: First Fitting Color, Decreasing Degree	

QAP Quadratic Assignment Problem	**Given:** Set of activities and locations along with the flows between activities and the distances between locations $d_{ij} \in D$:distance matrix $f_{ik} \in F$:flow matrix x_{ij}:1 if activity is assigned to location j, 0 otherwise **Goal**: assign each activity to a location to minimize total cost $\min z = \sum_{i,j=1}^{n} \sum_{k=1}^{n} d_{ij} f_{jk} x_{ij} x_{hk}$	
Neighborhoods	- swap 2 locations of 2 facilities $O(n^2)$ - swap locations of k=3,4,... facitilites $O(n^k)$	
Post office problems	**Goal**: Find the next post office (or ATM nowadays) in a city from your location. **Solution:** Use Voronoi diagram	
Exact cover	**Given:** Subset $U_i, i = 1..n$ of a base set M **Find**: Exact cover (if one exists), i.e. a choice of sets U_i such that their union is m and no element is contained in more than one of these sets. Optimization: Find a choice of sets U_i such that their union is M and as few element as possible are contained in U_i.	
n queens problem	**Goal:** Place n queens on a $n \times n$ board such that no two queens can capture each other.	

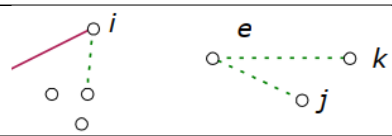
HEURISTICS

General

Heuristic	A heuristic method is based on knowledge acquired by experience on a given problem. They are opposed to exact algorithms. Heuristic methods do **not necessarily provide the best solution**, but need **reasonable time**.	
Meta-Heuristic	A meta-heuristic is a limited **set of concepts** that can be applied to a large set of combinatorial optimization problems and that allow creating new heuristic methods. It provides support for designing heuristic methods, based on knowledge acquired by designing heuristic method for various problems.	
Trajectory-based	Start with a solution and improve it continuously by exploring its "neighborhood". We obtain a trajectory through the solution space.	
Population-based	An evolving population of (partial) solutions, whose members evolve and adapt individually to the problem and are searching for the optimum. Problem specific information can be **exchanged** between the members of the population, and can be **passed on** to descendants.	
Neighborhood	Too small: get stuck quickly without exploring Too large: computation time to high	

Heuristics

For TSP:

Nearest Neighbour (=Greedy)	Start from city 1. Repeat: Go to the nearest city not yet visited, until all visited.	
Best Global Edge (=Greedy)	Initial S: $\emptyset$ R: Set of edges that can be added to S such that: No cycle is created, no vertex with degree > 2 is created $c(S,e)$: weight of edge e (this is independent from S)	
Maximum Regret (=Greedy)	$S = \{1\}$ R: Set of cities not yet visited $c(S,e)$ = Regret of not going to e from i Choose the largest $c(S,e)$	
Best Insertion (=Greedy)	S = Tour on 2 cities R: Set of cities not yet visited $c(S,e)$ = Minimum insertion cost of city e between 2 cities Choose the smallest $c(S,e)$	

For Vertex Coloring

First Fitting Color (=Greedy)	Select an ordering of the vertices $s = \emptyset$ // set of already colored vertices $R = V$ // set of vertices not colored yet $C(s,e)$ // set of colors that can be assigned to a vertex (sorted) Choose first color in $C(s,e)$	
Decreasing Degree (=Greedy)	sort the vertices by decreasing order of degree Remaining is identical to first fitting color	
D_{satur} (=Greedy)	choose the uncolored vertex with the highest "saturation degree", i.e. with the maximum number of different adjacent colors. Break ties by choosing the vertex with maximum degree.	

Meta Heuristics

Constructive

Random Sampling
"or Building"
	Idea: Generate a solution randomly, uniformly in the solution space
	Advantages: Simple, works good, easy to implement
	Disadvantages: bad solution quality, uniform distribution sometimes not trivial
	Sometimes not easy to find -> add penalty function
		`S = Random solution`
		`Repeat`
		`   Find another solution randomly`
		`   Use if better than S`
		`Until satisfied`

Greedy Construction	**Idea:** Build element by element, by adding systematically the most appropriate ("best") element.
	Needs a cost function that measures the quality of adding element e to a partial solution S.
	Adding an element generally implies restrictions for the next elements to add.
	Disadvantages: too short-sighted
	Works optimally for: MST, Shortest Path, ...
		R = E (set to add)
		Repeat
		   Evaluate $c(S, e)$ for each $e \in R$
		   Choose e' which optimizes $c(S, e)$
		   Add e' to the partial solution S
		   Remove from R all elements that cannot be added to S anymore.
		Until S is a complete solution

Exhaustive Search	**Idea:** Generate all feasible solutions and find the optimum **Advantages:** Guarantees to find an optimal solution Often easy to implement **Disadvantages**: The set of feasible solutions is often exponential -> takes to much time.	

Pilot Method
$O(n^2)$ for TSP
	Idea: Evaluate the quality of adding an element e to s by completing it to a full solution. The heuristic to complete a solution must be chosen, e.g. greedy + local search, ...
	Remarks: Time complexity is increased by the "pilot"
		`For all e that can be added to S`
		`   Calc s+e with heuristic "pilot"`
		`   Keep the best solution`

Beam Search	**Idea:** Similar to exhaustive search, but define a beam width (e.g. 2) and browse depth.
	beam width $= \infty \rightarrow exhaustive\ with\ BFS$
		`For each partial solution`
		`   expand up to k depth`
		`   keep the most promising`

Local Improvement

**Local Improvement
= Local Search
= Hill Climping**
	Idea: Start with a given solution (with a constructive method) and find improvements
	Neighbourhoods: Terminating: fixed number, no improvement after x steps, when a target is reached
		`Repeat`
		`Try to find an improved solution`
		`Perform if found`
		`While An improvement is found`

Selection	**First Improving Move** (takes the first move that improves the current solution)	`While improving solution`
		`s' = new solution`
		`if c(s') < c(s)`
		`s = s'`

	Best Improving Move (take the best move that improves the current solution)	`costOfBestNewSol = a`
		`for each move m in M(s)`
		`s' = new solution with m applied`
		`if c(s') < costOfBestNewSol`
		`costOfBestNewSol = c(s')`
		`bestMove = m`
		`apply bestMove to s`

Neighbourhood	**TSP**: search for intersections, 2/3/k-opt, or-opt (move a subchain of vertices somewhere else) -> use a doubled chained list **CVRP**: First fit, Random, Closest (customer, point of gravity, edge), lowest left capacity, change beginning **Knacksack**: shift, invert, transpose(swap)	**Connectivity**: A global optimum can be reached from any feasible solution **Low diameter**: Number of moves for linking any pair of solutions is small **Low ruggedness**: Limited number of local optima **Limited size**: Number of neighbour solutions is small **Easy to evaluate**: neighbour solution must be found without prohibitive computation
Var	Always take best solution from neighborhood, even if this decreases target function. (+) to escape from local optima (-) might go back and forth (run into a cylce) (-) not monotone increasing	

Tabu Search
(=Local Search)
	Idea: Similar to hill climbing, but with some memory. Try to avoid steps that go back to previously visited solutions, or that undo the effect of previous steps.
	Goal: Promote diversity of the solutions explored, in particular to reduce cyclic behaviour and to escape from local optima.
	Tabu list: store last move / store last n moves / ...
	Allow invalid solutions but **penalize** them (fix or variable).
	Tuning: Learn tabu value (incr/decr or rand)
		`Start with a random solution`
		`Repeat`
		`Find a better solution`
		`Consider steps that ar not tabu`
		`Perform if better`
		`and update tabu list`
		`Until criteria reached`

example

Iteration	Variable flipped	Current solution	Revenue	Vol	Tabu list
0	-	0 0 0 0 0 0 0 0 0 0		0	
1	4	0 0 0 1 0 0 0 0 0 0	12	14	4
2	1	1 0 0 1 0 0 0 0 0 0	12+11=23	14+33=47	1,4
3	2	1 1 0 1 0 0 0 0 0 0	23+10=33	47+27=74	1,2,4
4	3	1 1 1 1 0 0 0 0 0 0	33+9=42	74+16=90	1,2,3
5	4	1 1 0 0 1 0 0 0 0 0	43-		

ACO
Ant Colony Optimization
	Param:
	Distance matrix $D = (d_{ij}) \rightarrow given$
	Trace matrix $T = (\tau_{ij})$ →pheromone trail
	α: weight of the pheromone (0 → nearest neighbor)
	β: weight of the distance (0 → push very few tours)
	ρ:
	τ_0: init of the pheromone trail
	Q: constant for pheromon update
	m: batch size
	$\max_{iter}$: maximal iterations
		`Set parameters`
		`Initialize pheromone trails`
		`Do`
		`   Construct Ant Solutions`
		`   Apply local search (optional)`
		`   Update Pheromones`
		`While criteria reached`

e.g. for TSP

```
Data
   Distance matrix D = (d_ij) between cities
   Trace matrix T = (τ_ij)
   Parameters α, β, ρ, τ_0, Q, m, max_iter

Initialisation
   τ_ij = τ_0 for all i, j

Repeat for max_iter iterations
   R = (r_ij) = 0                                        // Edge reinforcement
   For each k = 1, …, m                                  // Ant k build a new solution
      L = 0                                              // Tour length
      Choose a city i at random                          // Current city : i
      Repeat, while all cities have not been visited
         Choose a city j not yet visited with probability proportional to τ_ij^α · d_ij^β
         L = L + d_ij
         i = j                                           Note: β ≤ 0

      For all arc (i, j) of the tour just built, set
         r_ij = r_ij + Q/L

   T = (1-ρ)T + R        // Update pheromone trail after having built m solutions

Return The best tour found
```

The shorter a tour of ant k, the higher the reinforcement value r_{ij} for the edges on this tour.

Var: **MMAS** MaxMin Ant System — one of the most successful

Changes:
Only the best ant updates the pheromone trails
Maintains lower and upper bounds τ_{min}, τ_{max} for pheromone values τ_e
Pheromone is updated for the "best" solution, for the current iteration or "best-so-far"
Update depends on the cost of the best solution

e.g. for TSP

```
Standard parameter settings
 α = 1, β = -2, ρ = 0.98, Q = 1, b = 20,  m = problem size
 τ_min = …, τ_max = …, max_iter = …                      // Problem dependent
Initialisation
   τ_ij = τ_max for all arc (i, j)

Repeat for max_iter iterations

   For each ant k = 1, …, m
      s_k = ∅                                            // Solution built by ant k
      Choose a city i at random                          // Current city : i
      Repeat, while all cities have not been visited
         If there is a city j not yet visited among b nearest neighbours of i
            Choose a city j among them with probability proportional to τ_ij^α · d_ij^β

         Else Choose a city j not yet visited with maximum τ_ij^α · d_ij^β
         s_k = s_k + (i, j)
         i = j
      Improve solution s_k with a local search

   For all arc (i, j) of the best tour s* among s_1, s_2, …, s_m
      τ_ij = max(τ_min , min(τ_max , (1 − ρ)τ_ij + Q/Length(s*) ) )

Return The best tour found
```

Important: There is no need that ants really follow a consecutive path -> like they can fly

Randomized Local Search

Noising Methods	Add a random noise when evaluating the moves with decreasing probability
	Typical Noise: standard deviation
	Generalisation of the next three
		`Repeat`
		`  Apply randomly move`
		`  Accept if it's better`
		`  Accept worse according to noise`

Var: **Threshold Accepting**	Accept all move deteriorating (=verschlechtern) the solution less than a given threshold	
Var: **Great Deluge**	Only accept solutions with a given minimal quality; increase level of quality (like a "flood")	

Var: **Simulated Annealing**	Similar to hill climbing, but allow non-improving moves.
	Probability: Fixed / decreasing over time / decreasing over time and depending on quality
	$\min\left\{1, e^{\frac{f(x_i)-f(y_i)}{T_i}}\right\}$
	If $f(x_i) \geq f(y_i)$ term gets ≥ 1 and y_i is accepted.
	If $T \to \infty$, every solution y_i will be accepted.
	If $T \to 0$, only better solutions will be accepted
	Advantages: spend more time on good solutions
		`Start with random solution`
		`Repeat`
		`  Apply randomly move`
		`  Accept if it's better`
		`  Accept with probability if worse`

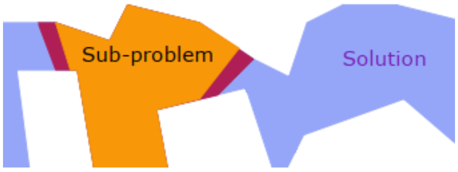
Restarting	When local search does not improve anymore, start a new local search from: - best solution so far (good solutions are close) - randomly generated (explore with large variety - GRASP) - modification of best solution (keep structure - VNS)	

Var: **GRASP** Greedy Randomized Adaptive Search Procedure	Input: s* best solution found so far	`Repeat`
		`  s = minimal partial solution`
		`  construct s' from s with greedy`
		`  find optima s'' in neighbourhood`
		`  if f(s'') < f(s*)`
		`    s* = s''`

Var: **VNS** Variable Neighbourhood Search	**Idea:** Working with p different neighbourhoods	`Repeat`
		`  Choose a randomly neighbourhood`
		`  find local optima in these`
		`  apply if better`

Decomposition Methods

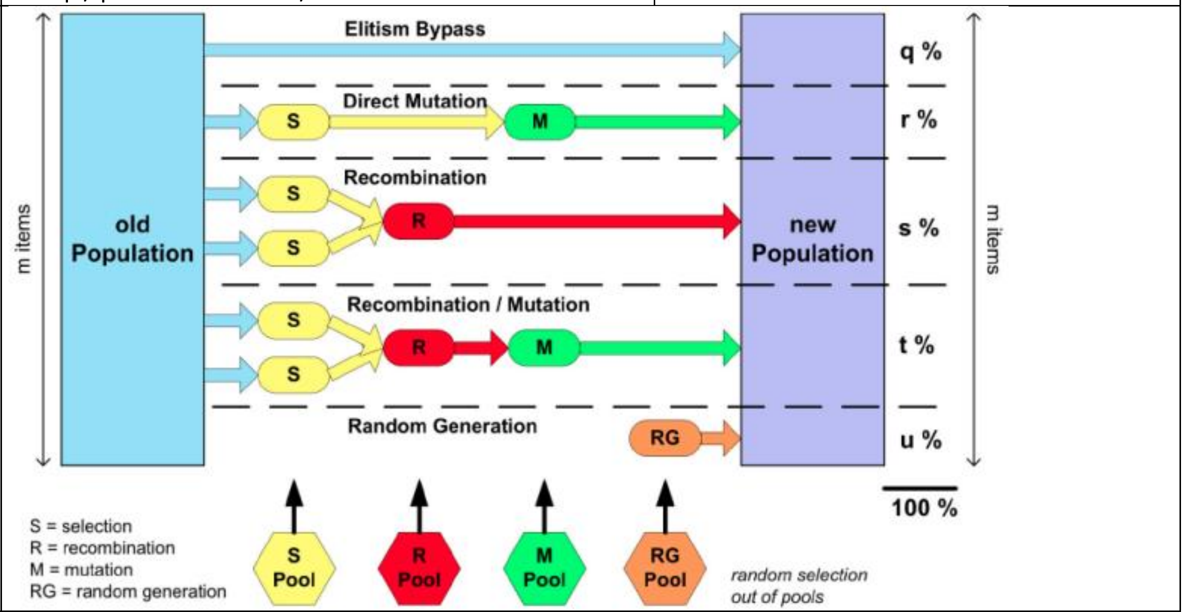
	Neighbourhood is too large to fully explore. Split problem in smaller problems, optimize local solutions	
VLNS Very Large Neighbourhood Search	**Idea**: Partially explore the large neighbourhood to find improvements **ILP**: fix value of subset and solve, repeat with other subset **Iterated Local Search**: randomly perturb (=stören) the best solution so far, and apply an improving method	

POPMUSIC
Partial Optimization Metaheuristic under special intensication conditions
	Idea: Decompose solution into parts, optimize several parts of the solution, repeat until optimized portions cover the entire solution space
	
	
	Difficulty: Sub-problems may not be independent from one another
		`Solution S=`$s_1 \cup s_2 \cup \ldots \cup s_p$
		$O = \emptyset$ `// already optimized parts`
		`Parameter r // "nearest parts"`
		
		`While` $O \neq S$
		  `Choose a seed part` $s_i \notin O$
		  `Create a subproblem R composed`
		    `of the r "closest" parts`
		  `Optimize subproblem R`
		  `If R improved then set` $O = O \setminus R$
		  `else` $O = O \cup s_i$

example	**Variables (example with VRP)** Part = Tour of one vehicle Distance = between centres of gravity Sub-Problem: A smaller VRP Optimization process: e.g. Tabu Search **Variables (example with TSP)** Part = a single city c on a tour Distance = r adjacent cities on the tour, immediately before/after c Sub-Problem: solve TSP for the r cities (e.g. exhaustive) Re-Combination: re-insert subtour in existing tour **Variables (example for permutation shop problem)** Part = a single job x Distance = take x as seed job and select r jobs that are scheduled before and after x Sub-Problem = Re-Schedule the selected job (use additional time constraints for start/end time -> still valid Re-Combination: shift all jobs as much to the left as possible	

<table>
<tr><td>Evolutionary Algorithms

Genetic Algorithms</td><td>Idea: Selection: Survival of the fittest (natural selection)
Finite lifespan of individuals (generations).
Inheritance of traits, info is passed on to descendants.
Recombination, info is exchanged between parents.
Diversification (by mutations).
Population acquires and cultivates shared knowledge (culture, collective memory)
Local development of populations (different cultures in different regions)</td><td>Choose a suitable encoding
Start at random population
Repeat
  Create next generation:
  - assign fitness to individuals
  - natural selection
  - choosing parents for reproduc.
  - recombination and mutation
Until criterion satisfied</td></tr>
<tr><td>Terminology</td><td>An individual is a possible solution candidate. e.g. vector
The genes are the individual entries of an individual.
The alleles are concrete values which a gene takes, e.g.0/1
The population is the set of all individuals at a given time.
A generation is the population at a specific point in time.
The genotype is the encoded form of an individual.
The phenotype is the decoded form of an individual, do not depend on encoding. Are the solution candidates.
The fitness function is our quality measure for a solution.</td><td></td></tr>
<tr><td>1. Encoding</td><td>Desirable: A small change in genotype corresponds to a small change in phenotype. -> use gray code
Example: Vector of n integer / binary vector / ...</td><td>standard: 00-01-10-11 (dist=1-2)
gray: 00-01-11-10 (dist=1)</td></tr>
<tr><td>2. Replacement schemes</td><td>General replacement: Replace all individuals
Strict elitism: Keep only m best individuals
Weak elitism: m best individuals are mutated
Delete-n: Replace n random individuals
Delete-n-last: Replace n worst individuals, keep others
Retirement home: Store some for later procreation</td><td></td></tr>
<tr><td>3. Selection</td><td>Selection pressure means that better individuals should have a higher chance of reproduction.
Strategy: Increasing selection pressure
- (Unbiased) Random selection
- Random selection with bias on better individuals (roulette wheel)
- Tournament selection</td><td>Better exploration when pressure is low
Better exploitation when pressure is high

Tournamen selection:<table><tr><td>best</td><td>2-nd</td><td>3rd-best</td><td></td><td>(k)-best</td></tr><tr><td>p</td><td>$(1-p)p$</td><td>$(1-p)^2p$</td><td>..</td><td>$(1-p)^{(k-1)}p$</td></tr></table>The larger k, the higher the selection pressure</td></tr>
<tr><td>4. Recombination</td><td>Crossing of individuals to exchange and pass advantageous properties.
- one-point crossover
- two-point crossover
- uniform crossover
- PMX (2-point with repair mechanism)
- adjacency method</td><td><table><tr><td></td><td>1-point</td><td>2-point</td><td>uniform</td></tr><tr><td>parent 1</td><td>1011|110</td><td>1011|000|001</td><td>10110</td></tr><tr><td>parent 2</td><td>0010|011</td><td>0010|110|100</td><td>00101</td></tr><tr><td>child 1</td><td>0010|110</td><td>1011|110|001</td><td>10111</td></tr><tr><td>child 2</td><td>1011|011</td><td>0010|000|100</td><td>00100</td></tr></table></td></tr>
<tr><td>5. Mutation</td><td>bit flip / position mutation / inversion</td><td></td></tr>
<tr><td></td><td colspan="2">
</td></tr>
</table>