

Evolutionary computation

Andrzej Jaskiewicz

Scope of the course

- Introduction
 - Elements of the optimization tasks. Sources of difficulty of optimization tasks. Examples of optimization problems with the focus on discrete/combinatorial problems. Classification of optimization methods
- Exhaustive search. The idea of the branch-and-bound method. Random search
- Greedy construction heuristics
 - The idea. Examples. Randomization. Regret heuristics
- Local search
 - The idea of neighborhood. Steepest and greedy local search. Improving efficiency of local search
- Extensions of local search
 - Multiple start local search. Variable neighborhood local search. Iterated local search. Adaptive local search. Large scale neighborhood search. Simulated annealing. Tabu search. Long term memory. Hyper-heuristics

Scope of the course

- Population-based and biologically inspired algorithms
 - Examples of methods. Genetic algorithms. Evolutionary algorithms. Crossover and recombination. The concept and role of schemata. Selection methods. Solutions encoding. Intermediate encoding. Hybrid evolutionary algorithms. Genetic hyper-heuristics. Ant colonies algorithms
- Constraints handling
- Theoretical basis
 - No free lunch theorem. Measures of difficulty of optimization tasks. Objective Function Landscape Analysis. Designing EAs/MHs for specific problems. Design patterns. Examples
- Experimental evaluation of metaheuristics
- Multiobjective metaheuristic and evolutionary methods
- Recent trends and research

Recent experience of the lecturer

<https://www.sintef.no/projectweb/top/vrptw/1000-customers/>

You are here: TOP / VRPTW / 1000 customers

NEARP / MCGRP

PDPTW

▼ VRPTW

Documentation

Solomon benchmark

25 customers

50 customers

100 customers

Gehring & Homberger benchmark

200 customers

400 customers

600 customers

800 customers

1000 customers

Contact information

1000 customers

Here you find instance definitions and the customer instances of Gehring & Homberger. It has a hierarchical objective: 1) Minimum Distance and time should be calculated with two decimals. Exact methods typically use precision distance and time calculations. It

Instance definitions (text)

Here you find [a zip file with the 1000 customer instances](#)

Best known results for Gehring & Homberger instances

The instance names in blue are hyperlinks that have all been checked by our solution checker against a peer reviewed publication. For each instance we provide the best known solution, the computing time, and the experimental platform used. We guarantee that the solutions have been properly checked and detailed solutions published earlier. We mention 'restricted', 'freestyle', and 'freestyle', the latter with no restrictions.

Instance	Vehicles
c1_10_1q	100
c1_10_2	90
c1_10_3	90
c1_10_4	90
c1_10_5i	100
c1_10_6	99
c1_10_7	97
c1_10_8	92
c1_10_9	88

Q - Quintiq. <http://www.quintiq.com/optimization/vrptw-world-records.html>

~~RP: S. Ropke & D. Pisinger. "A general heuristic for vehicle routing problems".
Department of Computer Science, University of Copenhagen.~~

SCR - Piotr Sielski (piotr.sielski@wmii.uni.lodz.pl), Piotr Cybula, Marek Rogalski (marek.rogalski@wmii.uni.lodz.pl), Mariusz Kok, Piotr Beling, Andrzej Emapa S.A. (<http://www.emapa.pl>), "New methods of VRP problem optimization", project funded by The National Centre for Research and Development, project number 0222/16.

WA - Ruud Wagemaker. MSc thesis in progress. Tilburg University.

Elements of an optimization tasks

- Optimization goal(s) – objective function and optimization direction – minimization/maximization
 - Multiple goals in multiobjective optimization
- Decision space
- Constraints on feasible decisions
- How to assess the impact of decisions on the optimization goal – how the objective function is calculated

Formulation of an optimization task

- Declarative approach:
 1. What goal do we want to achieve?
 2. What decisions can we make?
 3. What constraints do we need to take into account?
 4. How to calculate the impact of our decisions on our goal?

Example – vehicle routing

1. Goal: to minimize the costs of delivering goods to customers
2. Possible decisions: assignment of customers to vehicles order of customer services/visits
3. Constraints: vehicle capacity vehicle/driver time windows customer time windows...
4. Calculation of the objective function: analytical linear function

Development of optimization methods in recent years

- Significant development of the methodology allowing to solve more and more complex and larger tasks
- Integration with IT systems
- Multi-level relationships with artificial intelligence – theoretical foundations optimization in AI AI in optimization

Formulation of an optimization task

minimize/maximize $z = f(\mathbf{x})$

subject to constraints (s.t.)

$$\mathbf{x} \in S$$

minimize $z = f(\mathbf{x})$

is equivalent to

maximize $z' = -f(\mathbf{x})$

Optimal solution (optimum)

For minimization solution $\mathbf{x}^{opt} \in S$ such that

$$f(\mathbf{x}^{opt}) \leq f(\mathbf{x}) \text{ for each } \mathbf{x} \in S$$

For maximization solution $\mathbf{x}^{opt} \in S$ such that

$$f(\mathbf{x}^{opt}) \geq f(\mathbf{x}) \text{ for each } \mathbf{x} \in S$$

Also called a **globally** optimal solution although this adjective is formally unnecessary

It does not have to be unique if more solutions meet the above condition

Mathematical programming

- An optimization task can be defined as a mathematical programming task if:

the solution is defined as a vector of variables:

$$\mathbf{x} \in R^n \quad \mathbf{x} = \{x_1 \dots x_n\}$$

and constraints as a set of mathematical equalities/inequalities:

$$\mathbf{x} \in S \Leftrightarrow g_j(\mathbf{x}) \leq / \geq / = 0 \quad j=1 \dots L$$

Classification of mathematical programming task

- Objective function and constraints are linear \Rightarrow
 - linear mathematical programming task
- The objective function or one or more of constraints are nonlinear \Rightarrow
 - non-linear mathematical programming task
- Continuous variables \Rightarrow
 - continuous mathematical programming task
- Discrete variables \Rightarrow
 - discrete mathematical programming task
- Mixed continuous and discrete variables \Rightarrow
 - mixed discrete mathematical programming task
- Integer variables \Rightarrow
 - integer mathematical programming task
- Binary variables \Rightarrow
 - binary mathematical programming task

Various combinations possible e.g. integer linear mathematical programming binary nonlinear mathematical programming...

Mathematical programming solvers

- Mathematical programming tasks may be solved using standard solvers
- Effective solvers exist mainly for:
 - linear mathematical programming (LP)
 - mixed linear mathematical programming(MIP)
 - eg. Gurobi Cplex Frontline Lindo Matlab...

Mathematical programming solvers - development in recent years

- Interior point method
 - Inspired by nonlinear programming methods
 - Better than the simplex method for large problems
- Preprocessing – removal of redundant (always inactive) constraints and variables
 - e.g.: $x_1 + x_2 \leq 5$, $x_2 + x_3 \leq 5 \Rightarrow x_1 + x_3 \leq 5$
- Branch and price – combination of branch and bound with column generation
- Branch and cut – combination of branch and bound with cutting planes
- Matheuristics – combination of MP solvers with metaheuristics
- Optimization of calculations (e.g. parallelism)

The importance of modeling

- E.g. Chebycheff (scalarizing) function
 - minimize $\max \left(\lambda_j \left(z_j^0 - f_j(x) \right) \right) \quad j = 1 \dots p$
 - s.t.
 - $x \in S$
- Linear version:
 - minimize ε
 - s.t.
 - $\varepsilon \geq \lambda_j \left(z_j^0 - f_j(x) \right) \quad j = 1 \dots p$
 - $x \in S$

Combinatorial optimization tasks and problems

- A discrete and finite set of solutions
- Combinatorial structure (sets permutations combinations trees graphs...)
- A combinatorial optimization task can always be defined as a mathematical programming task (discrete binary). It is often associated with a rapid increase in the number of variables and constraints (e.g. n^2 variables for the travelling salesperson problem)
- This allows the use of universal mathematical programming solvers but in general is not necessary when using metaheuristic/evolutionary methods.

Concepts of the combinatorial optimization problem and instance

- The combinatorial optimization problem is a set of instances defined according to a certain scheme
- Instance is specific optimization task - goal function decision space constraints direction of optimization
- An instance is created by determining certain parameters of the problem e.g. the number of vertices and the distance between vertices in the traveling salesperson problem

Examples of combinatorial optimization problems

- Assignment problem
- Knapsack problem
- Minimum spanning tree problem
- Set covering problem
- Traveling salesperson problem - TSP
- Vehicle routing problem - VRP
- Graph coloring problem
- Clustering
- ...

Assignment problem

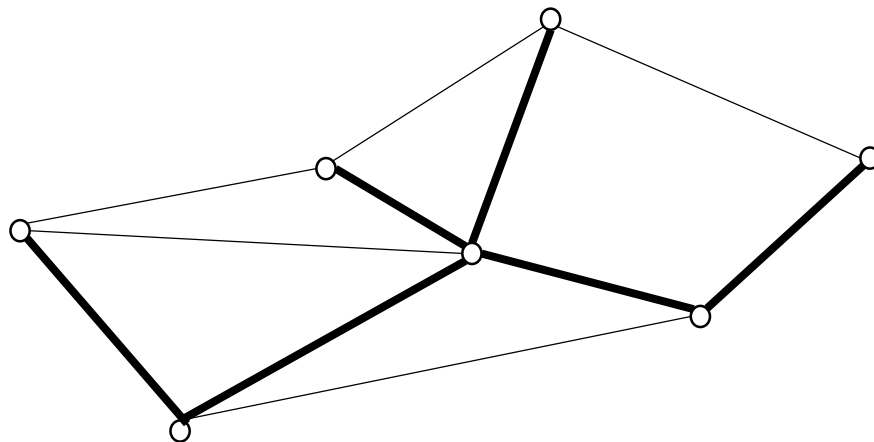
- A set of items and tasks is given. Each item can be assigned to each task. Assigning each item to a task comes at a cost. You can assign one item to each task. Find an assignment of items to tasks that minimizes the total cost.
- For example assigning employees to tasks

Knapsack problem

- A set of items of a given weight and value is given. Select the items to be placed in the knapsack so as not to exceed the capacity of the knapsack and maximize the total value of the selected items
- Eg.
 - A thief choosing phantoms to take
 - Investor choosing the components of the investment portfolio

Minimum spanning tree problem

- Find the spanning tree (the set of edges connecting all vertices/nodes) in a weighted graph with a minimum total weight
- For example planning telecommunication connections with a minimum total cost connecting all vertices

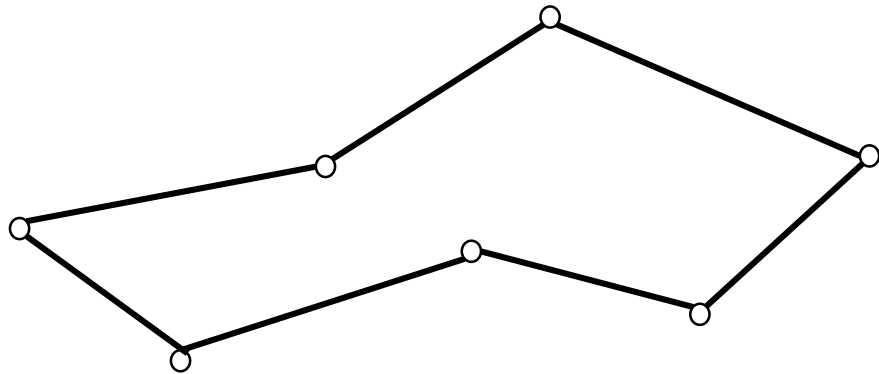


Set covering problem

- A set of elements and a set of subsets of these elements are given. Each subset has a certain cost. Cover all elements by choosing subsets with the minimum total cost.

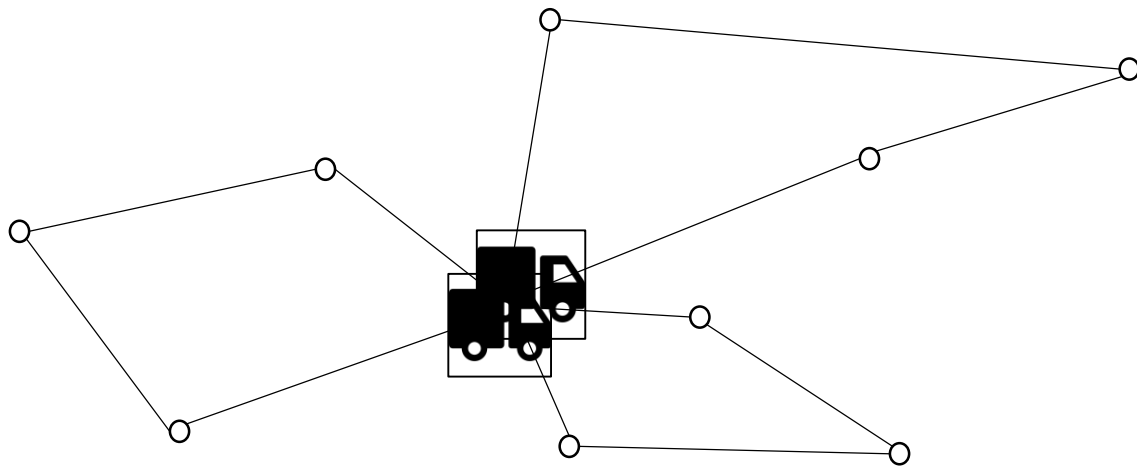
Traveling salesperson problem – TSP

- A set of vertices/nodes (cities) and distances between each pair of vertices (full weighted graph) are given. Find the Hamiltonian cycle (a closed path passing through all vertices) with a minimum total length
- Eg.:
 - A common element of practical transport problems
 - Planning the sequence of robot operations



(Family of) vehicle routing problems

- A set of vertices is given along with the distances and parameters of the vehicle fleet (e.g. capacity time windows) including the base vertex. Find a set of routes visiting all vertices starting and ending at the base meeting all restrictions with a minimum total length or the number of routes



Graph coloring problem

- Color (assign labels) to the vertices of the incomplete graph so that vertices of the same color are not connected by an edge using as few colors as possible
- E.g. frequency allocation to antennas in cellular networks (edge indicates the risk of interference)

Clustering

- Cluster a set of elements for which a certain measure of similarity or distance is known minimizing a certain measure of clustering quality e.g. the ratio of the average distance of solutions placed in one cluster to the average distance of solutions placed in different clusters

Versions of the problems

- Virtually every problem may have different versions
- For example for TSP and VRP one can consider:
 - Time windows
 - Vehicles'/drivers' working times
 - Precedence constraints
 - Heterogeneous or homogeneous vehicle fleet
 - Directed or undirected graphs – symmetric or asymmetric problems
 - ...

Application areas for combinatorial optimization

- transport
- logistics
- production planning
- project and staff scheduling
- robotics
- telecommunications
- artificial intelligence and machine learning
- engineering design
- medicine
- resource optimization (including IT)
- software engineering
- marketing
- finance/investments...

Sources of difficulty of optimization tasks/problems

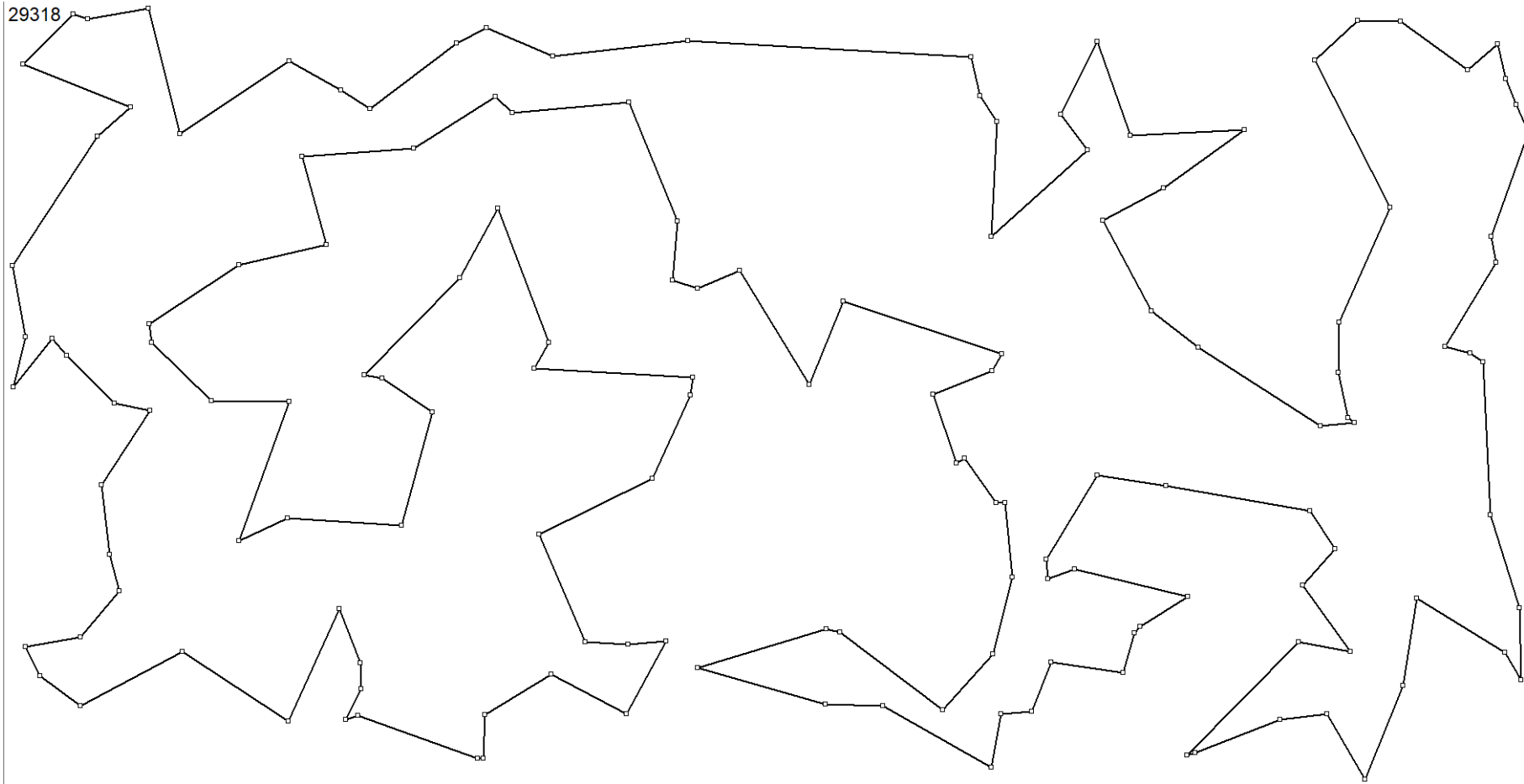
- Large number of solutions
- Constraints
- Computational complexity
- Uncertainty (not considered within the course)

Large number of solutions

- Knapsack problem with n elements:
 - 2^n subsets although fewer feasible solutions and fewer solutions satisfying the optimality condition (i.e. no possibility to add an element to the knapsack)
 - Eg. $n = 100$ to $127 * 10^{30}$ solutions
- Assignment problem with n items and m tasks
 - Can be encoded as a matrix $n \times m$ with the choice of individual allocations so 2^{nm} although fewer feasible solutions
- The symmetric traveling salesperson problem with n nodes
 - The number of permutations $n - 1$ elements (the choice of the first one does not matter) $(n - 1)!$
 - Eg. $n = 20$ $24 * 10^{18}$ solutions $n = 100$ $93 * 10^{157}$ solutions
- On the other hand an infinite number of solutions in continuous problems can make them easier to solve

Travelling salesperson problem with 150 nodes

- Exemplary solution



Can we solve such problem?

- Number of solutions $(n-1)!$
- $(150-1)! = 38 \times 10^{260}$
- The Age of the Universe in Planck's Units of Time $\approx 10^{61}$
- Particle number in the observable Universe $\approx 10^{80}$
- So if every particle were a computer and generated one solution in Planck time we could generate so far $\approx 10^{141}$ solutions
- Definitely we cannot solve this problem by exhaustive search

Constraints

- Theoretically they make the task easier because they reduce the number of acceptable solutions
- In practice they often make the task harder
- In particular finding a feasible solution may be very difficult
- It is also becoming more difficult to move between solutions as intermediate solutions may be infeasible

Computational complexity

- Problems from class P – known effective algorithms with polynomial complexity
 - E.g. the assignment problem the minimum spanning tree problem
- Problems from the NP class – only algorithms of exponential complexity are known - but easy (polynomial time) to verify a given solution
- Open problem if $P \neq NP$
- NP-hard problems. Any other problem from the NP class can be transformed into an NP-hard problem in polynomial time. NP-complete if they are simultaneously in the NP class. E.g. knapsack problem TSP graph coloring
- Practical notes:
 - Typically computational complexity concerns the worst case (instance) (analysis and even definition of the average case is much more difficult). In practice problems even NP-hard can be very different in behavior in the average case. For some problems most instances can be solved quite easily (e.g. knapsack problem TSP) for other problems most instances require exponential time (e.g. graph coloring)
 - Polynomial complexity can still be problematic for higher-degree polynomials

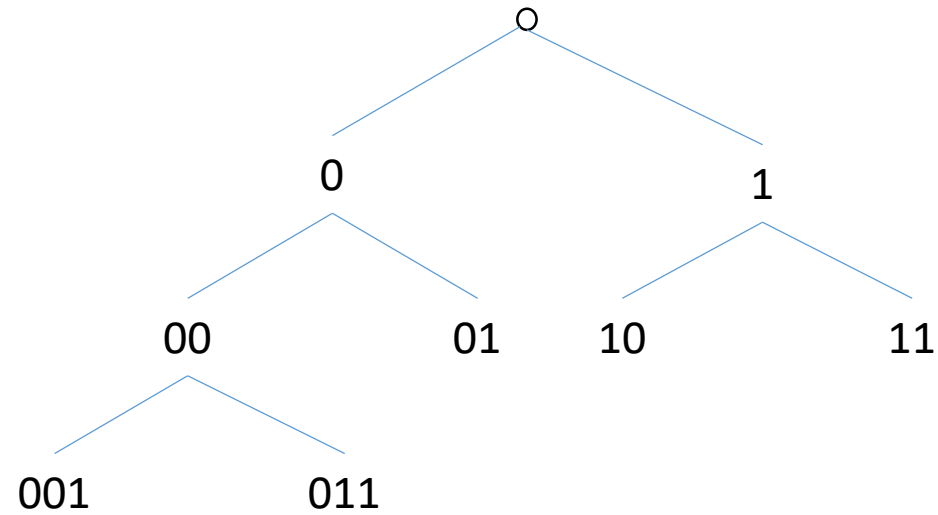
Classification of optimization methods

- Exact methods that guarantee finding the (global) optimum
 - Exhaustive search branch and bound methods mathematical programming methods (and solvers) dedicated methods – usually some versions of B&B dynamic programming
- Approximate methods – heuristics
 - Random search and walk dedicated heuristics greedy construction heuristics metaheuristics exact methods with limited time
 - They may or may not give some guarantee of approximation
- Can also be divided into general methods (universal applicable to various problems) and methods dedicated to specific problems although the division is not sharp. Dedicated methods are usually some adaptations of general schemes

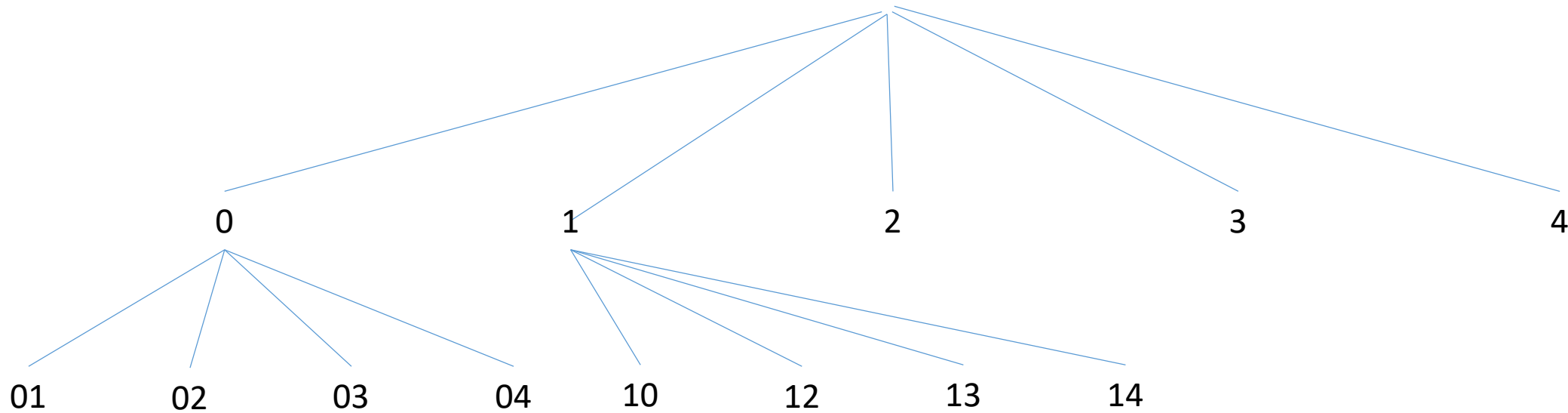
Exhaustive search

- We generate and calculate the value of the objective function for all solutions
- Difficult or practically impossible with a large number of solutions
- A systematic way to generate all solutions is needed (preferably without repetitions)
- E.g. for binary strings (assignment problem knapsack problem) decoding consecutive integers
 - 0 000
 - 1 001
 - 2 010
 - 3 011
 - 4 100 (is it possible to avoid changing more than one digit?)
- Often organized as passing a tree

Exhaustive search tree for binary strings



Exhaustive search tree for permutations (e.g. traveling salesperson problem)



Branch-and-bound method

- Modification of the exhaustive search organized as passing a tree
- Before expanding a branch the value of the lower/upper bound for the optimal value of the objective function in that branch is calculated – we assume that this can be done much faster than expanding the entire branch
- If the lower/upper bound is worse than the best value already found then the branch is not expanded (there is certainly no better solution in it)
- How to find the lower/upper bound
 - For example for a knapsack problem one can quickly find the optimum for a relaxed problem (where one can choose fractional parts of items) which gives the upper bound for the original binary problem
 - For example for a traveling salesperson problem the lower bound is given by the minimum spanning tree
- The order of the branches can be of great importance for time efficiency, the best approach is to quickly improve the best solution
- If we interrupt before expanding all the uncut branches the method becomes a heuristic. Then the order of the branches also affects the quality of solutions.

The complexity of black-box search on quantum computers

- On classical computers $O(|S|)$ – the only way out is an exhaustive search
- On quantum computers using Grover algorithm extensions $\Theta(\sqrt{|S|})$
 - Provably there is no better quantum algorithm
 - Complexity remains exponential for the exponential size of the solution space (so this does not mean that quantum computers can effectively solve NP-complete problems)
 - NP-complete problems could be solved in polynomial time if quantum mechanics were at least slightly nonlinear
 - Is the inability to effectively solve NP-complete problems a law of nature?
 - Daniel S. Abrams and Seth Lloyd. 1998. Nonlinear Quantum Mechanics Implies Polynomial-Time Solution for NP-Complete and # P Problems. Phys. Rev. Lett. 81 (Nov 1998) 3992–3995. Issue 18. <https://doi.org/10.1103/PhysRevLett.81.3992>
 - Scott Aaronson. 2005. Guest Column: NP-Complete Problems and Physical Reality. SIGACT News 36 1 (March 2005) 30–52. <https://doi.org/10.1145/1052796.1052804>

Random search

repeat

 generate and evaluate (calculate the value of the objective function)
 a random solution

until the stopping conditions are met

return the best solution found

Random walk

generate and evaluate a random solution

repeat

 randomly modify and evaluate the current solution

until the stopping conditions are met

return the best solution found

- Random modification may be faster than generating a random solution from scratch
- Sampling of the solutions space is less uniform

Construction heuristics/methods

- The solution to the optimization problem is constructed in steps by adding more elements (or in general other one-way operations e.g. joining routes in the Clarke-Wright heuristic (for VRP))
- Eg.
 - Adding items to an incomplete solution to a knapsack problem
 - Adding vertices/edges to an incomplete solution to the traveling salesperson problem
- If subsequent elements are selected randomly this gives a way to create a random solution

Greedy construction heuristics

- Arise if subsequent elements are selected to optimize the current change in the value of the objective function
- Eg.
 - Add the best value-to-weight item to your knapsack
 - Add the closest vertex to an incomplete solution of the traveling salesperson problem

Greedy construction heuristics

Create an incomplete initial solution eg. $\mathbf{x} := \emptyset$

repeat

 add to \mathbf{x} the currently best element not yet included in the solution

until a full solution is created

General greedy construction heuristics

Create an (incomplete) initial solution

repeat

 perform the currently best available one-way operation

until one-way operations are available

The nearest neighbor construction heuristics for TSP

select (e.g. randomly) the starting vertex

repeat

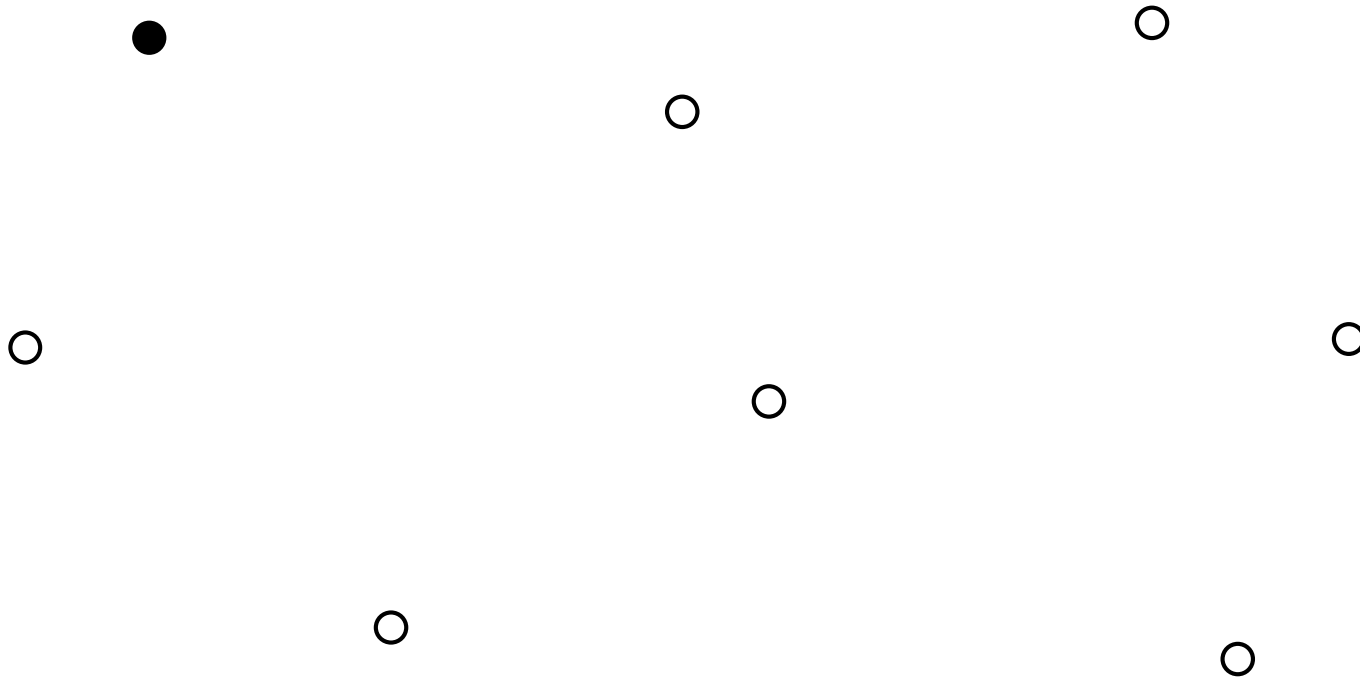
 add to the solution the vertex (and the leading edge) closest to the last one added

until all vertices have been added

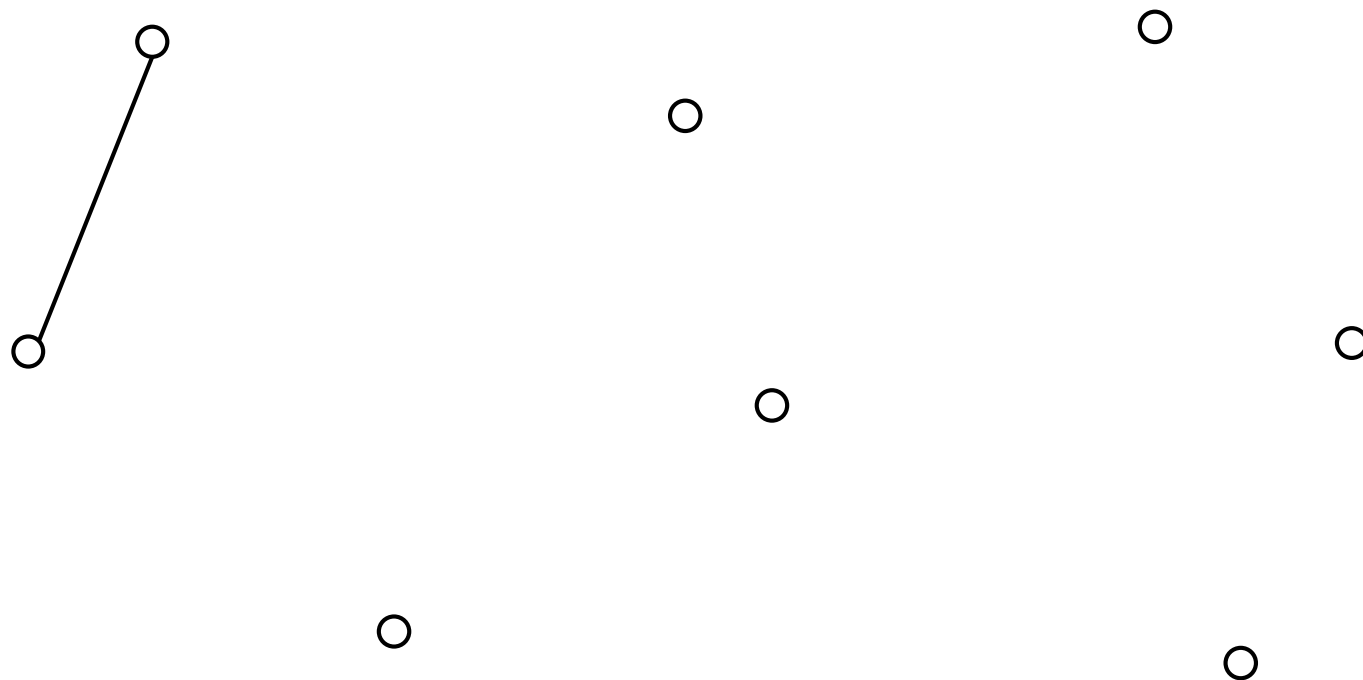
Add the edge from the last to the first vertex

There is also another version which considers all possible places of insertion of subsequent vertices (not only at the end)

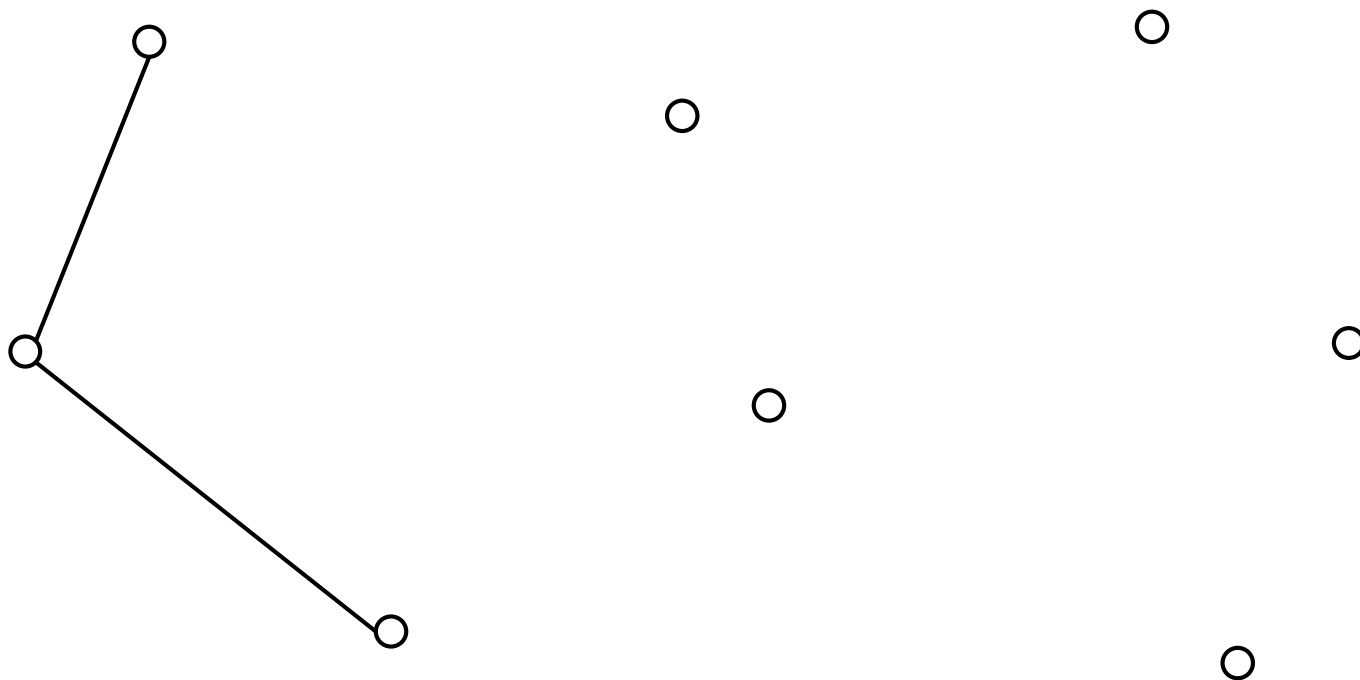
The nearest neighbor construction heuristics for TSP



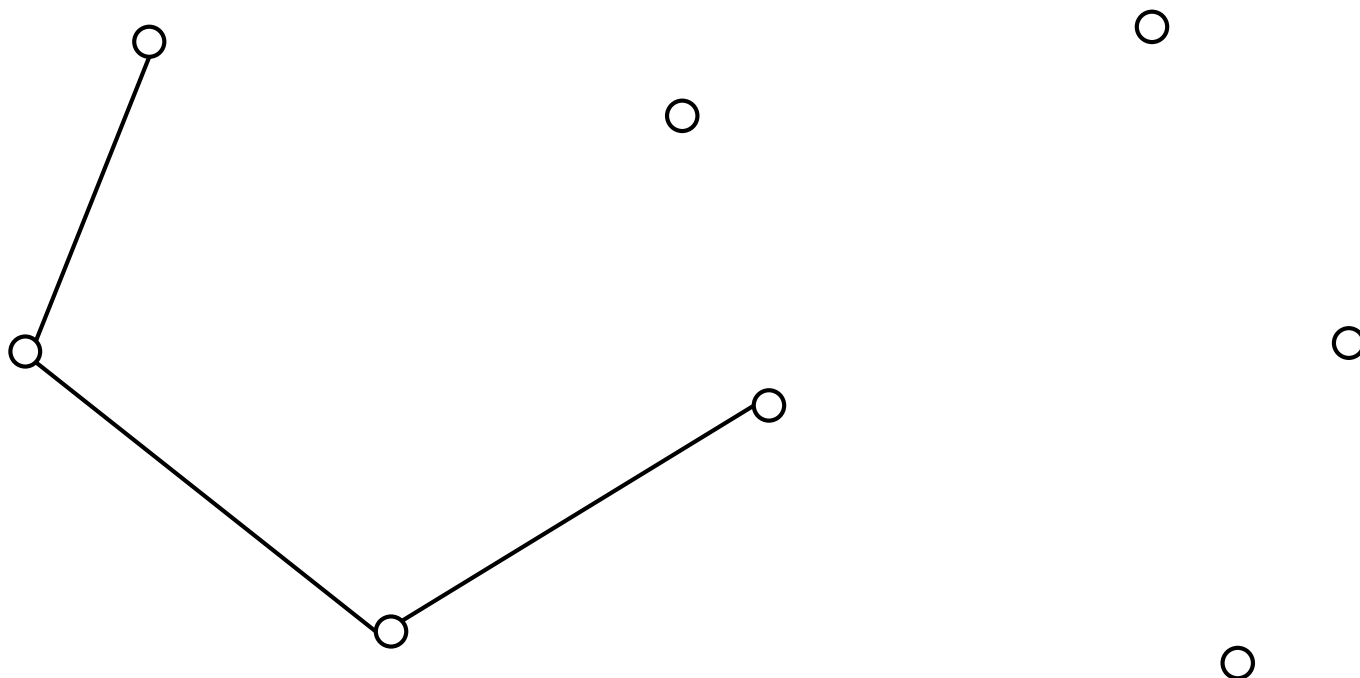
The nearest neighbor construction heuristics for TSP



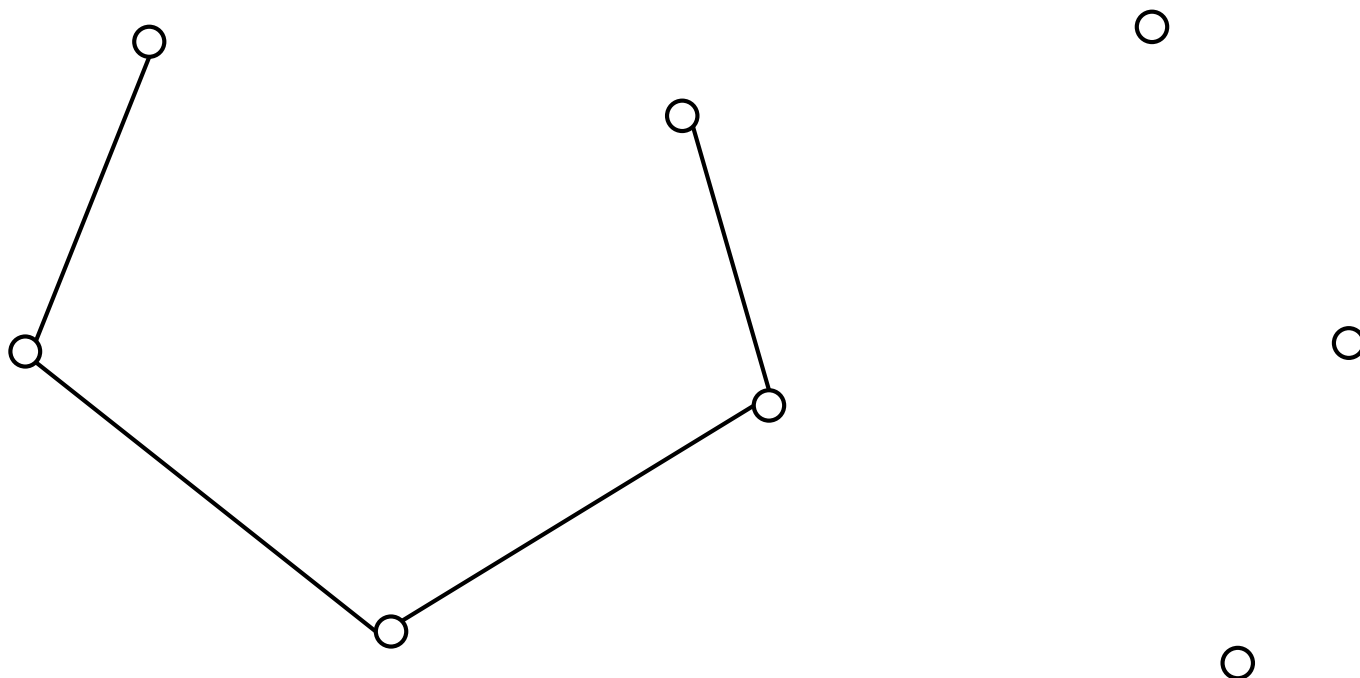
The nearest neighbor construction heuristics for TSP



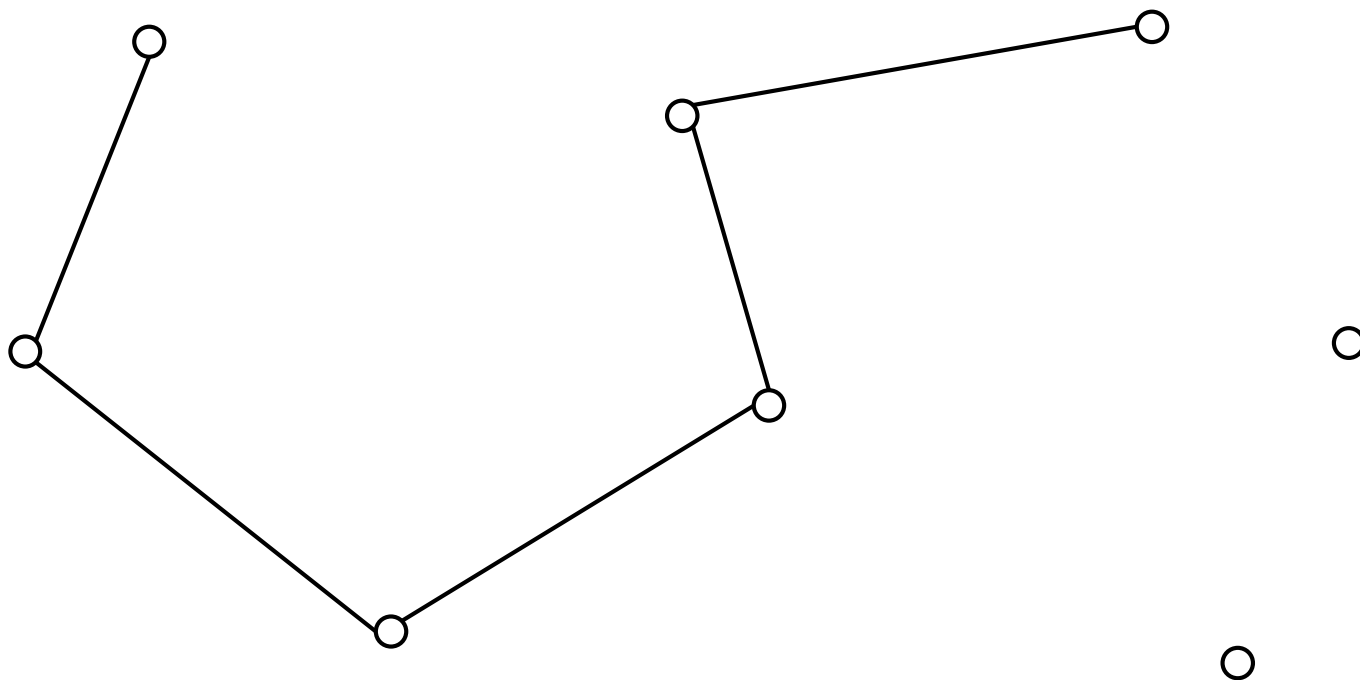
The nearest neighbor construction heuristics for TSP



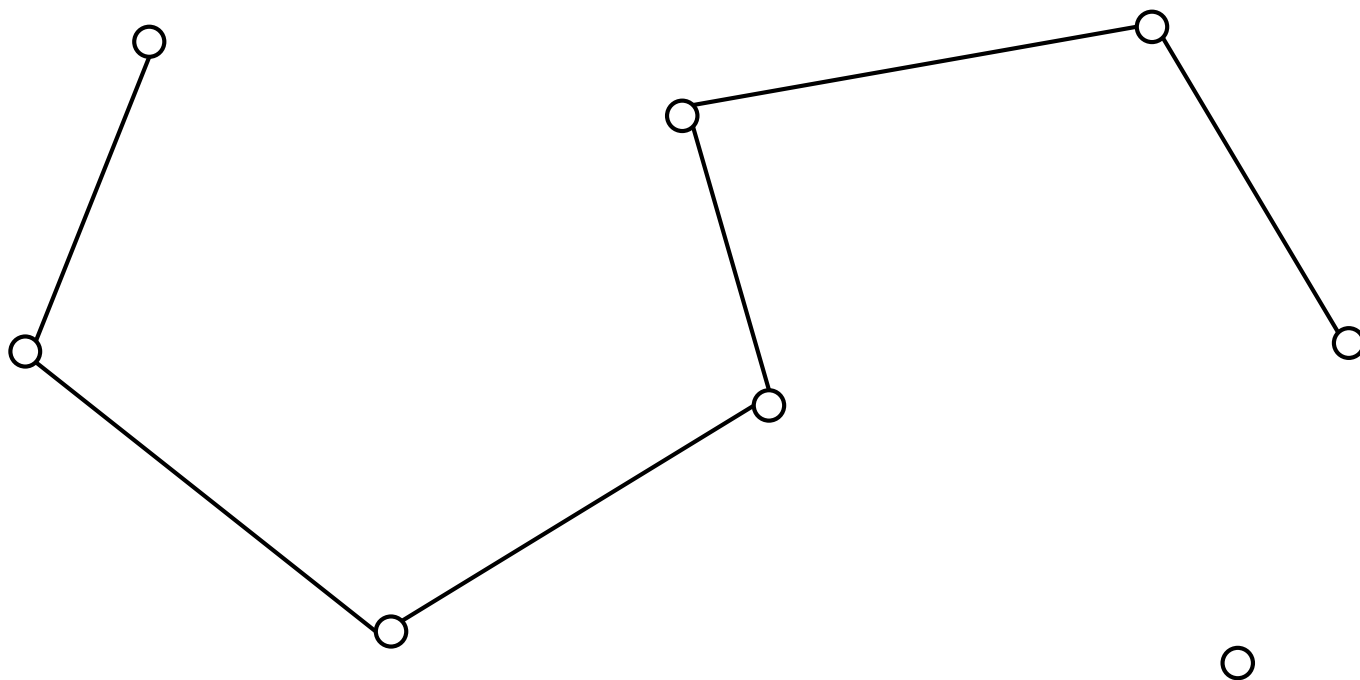
The nearest neighbor construction heuristics for TSP



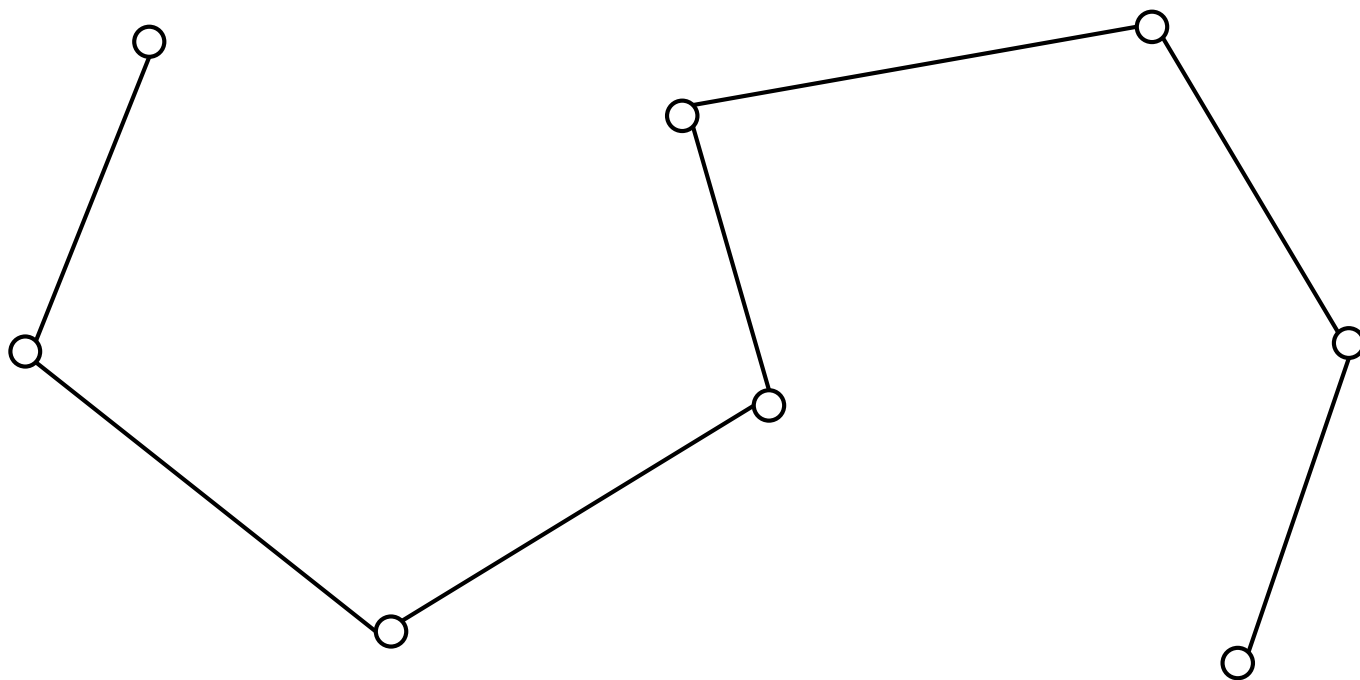
The nearest neighbor construction heuristics for TSP



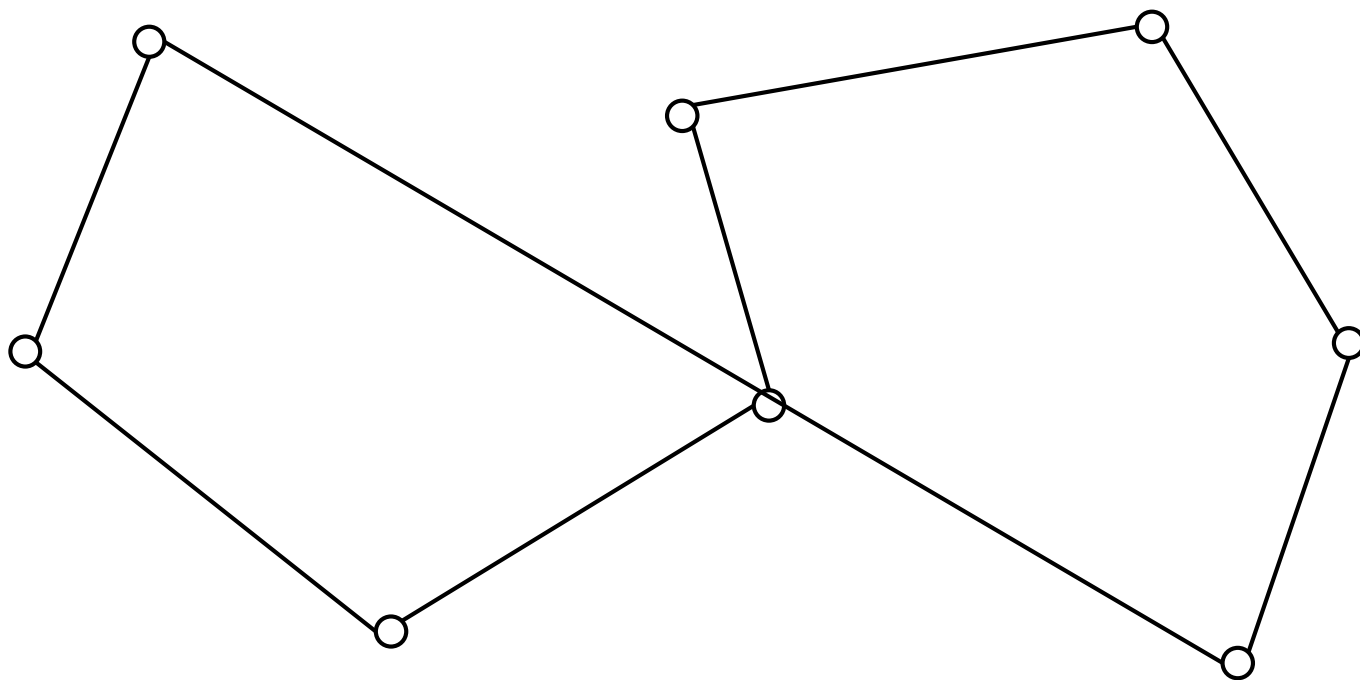
The nearest neighbor construction heuristics for TSP



The nearest neighbor construction heuristics for TSP



The nearest neighbor construction heuristics for TSP



Greedy cycle construction heuristics for TSP

select (e.g. randomly) the starting vertex

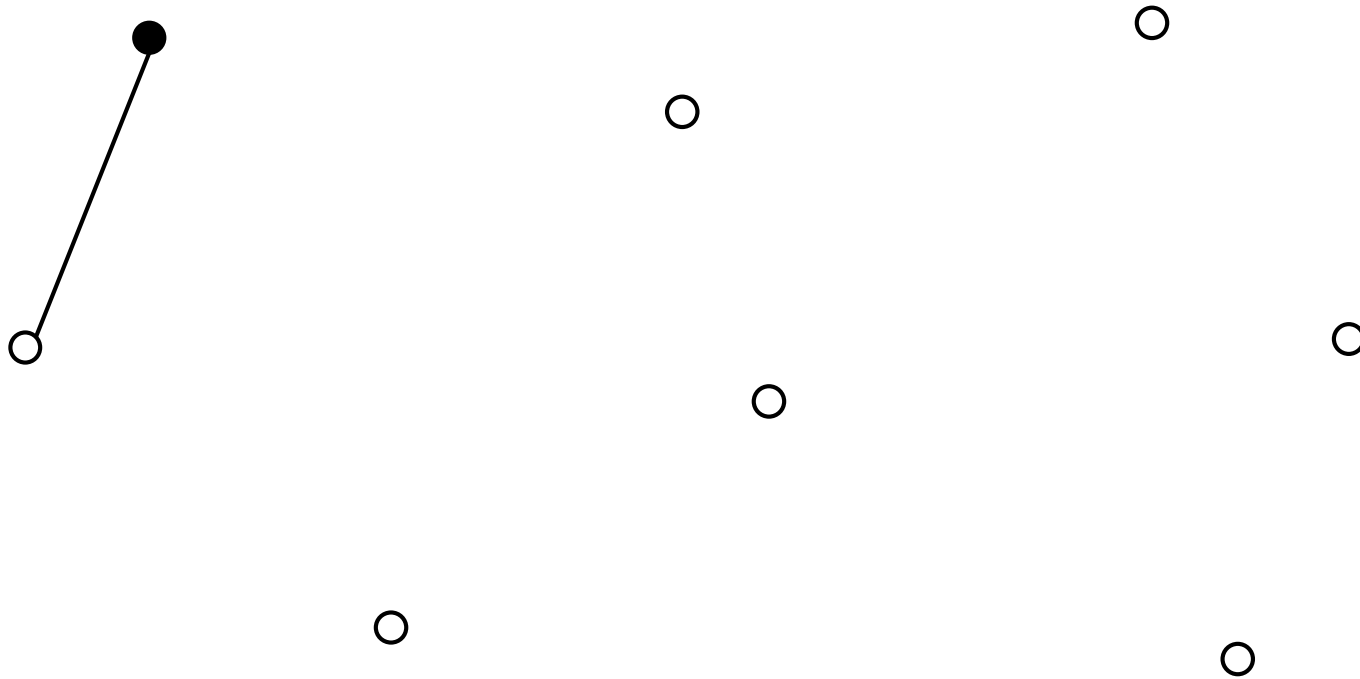
choose the nearest vertex and create an incomplete cycle from these two vertices

repeat

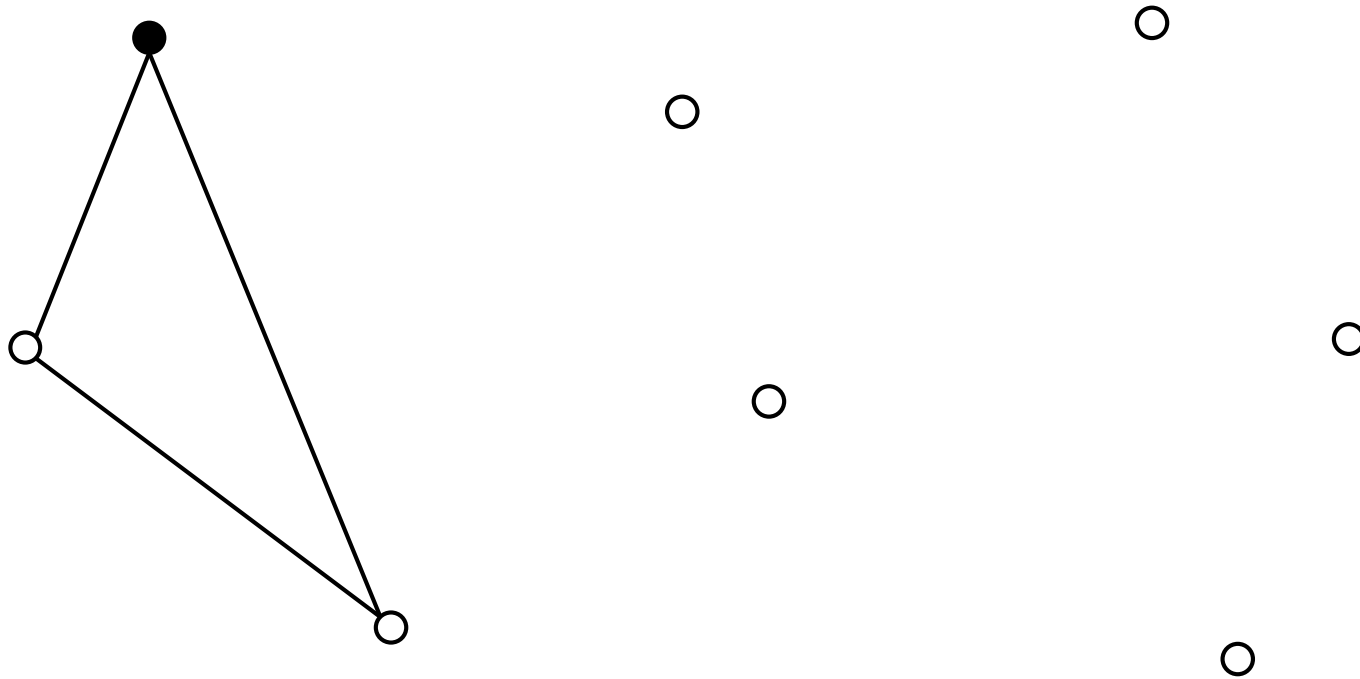
 insert into the current cycle in the best possible place the vertex
 causing the smallest increase in cycle length

until all vertices have been added

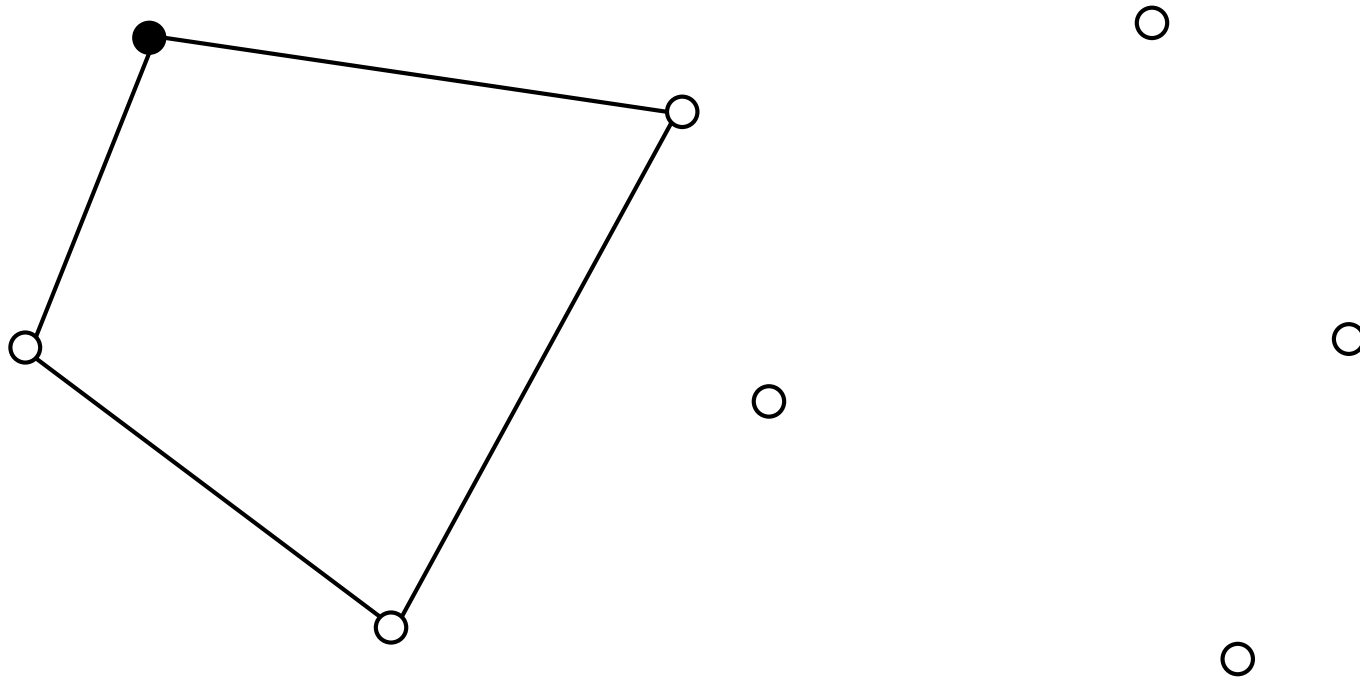
Greedy cycle construction heuristics for TSP



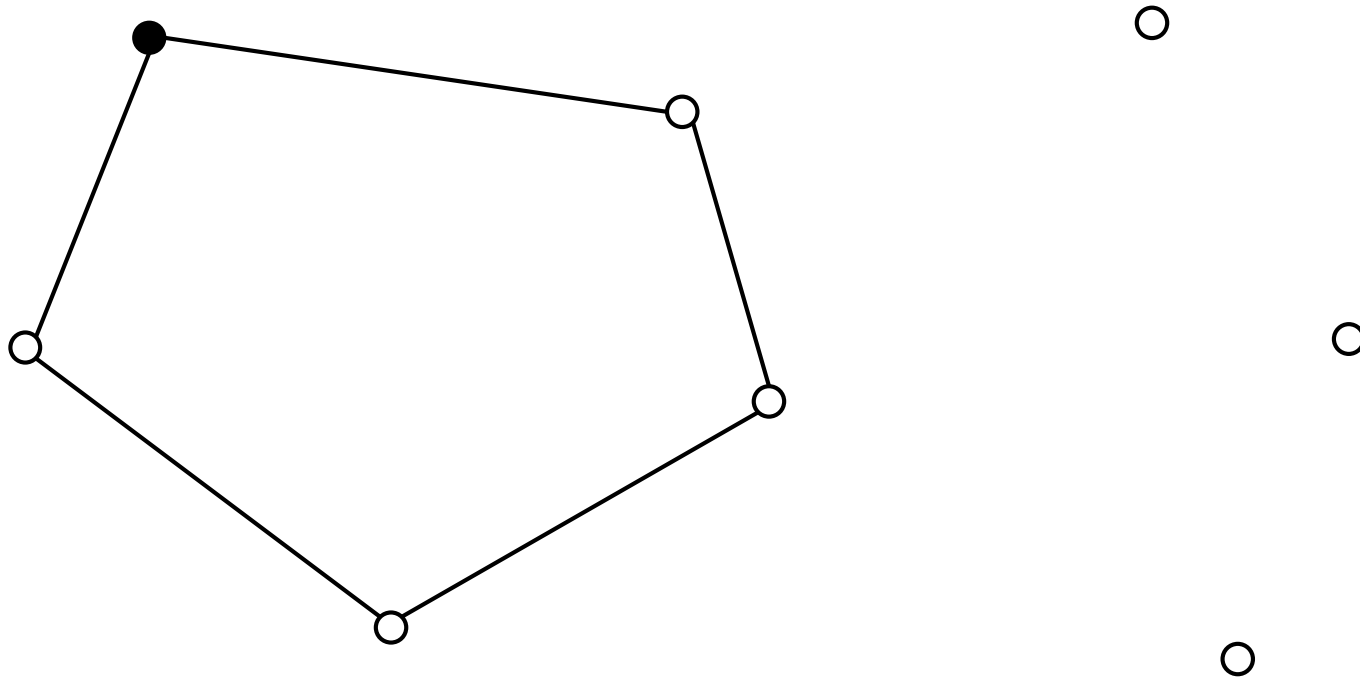
Greedy cycle construction heuristics for TSP



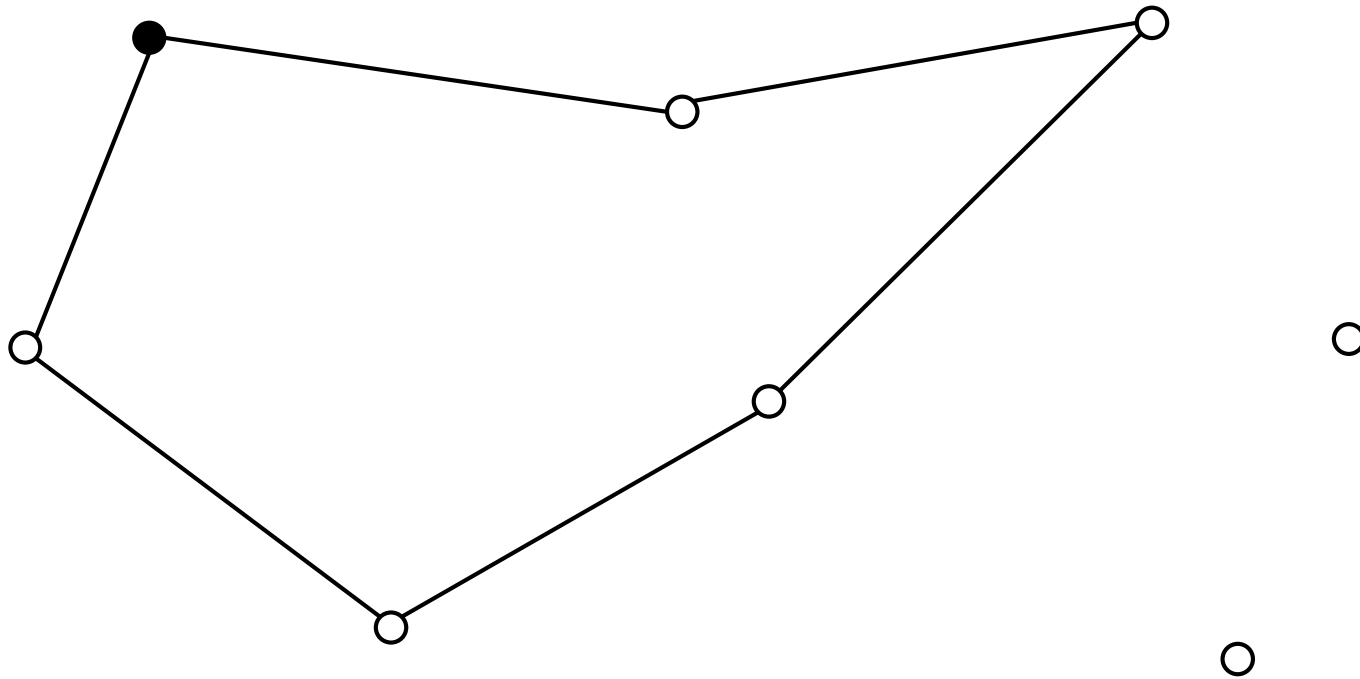
Greedy cycle construction heuristics for TSP



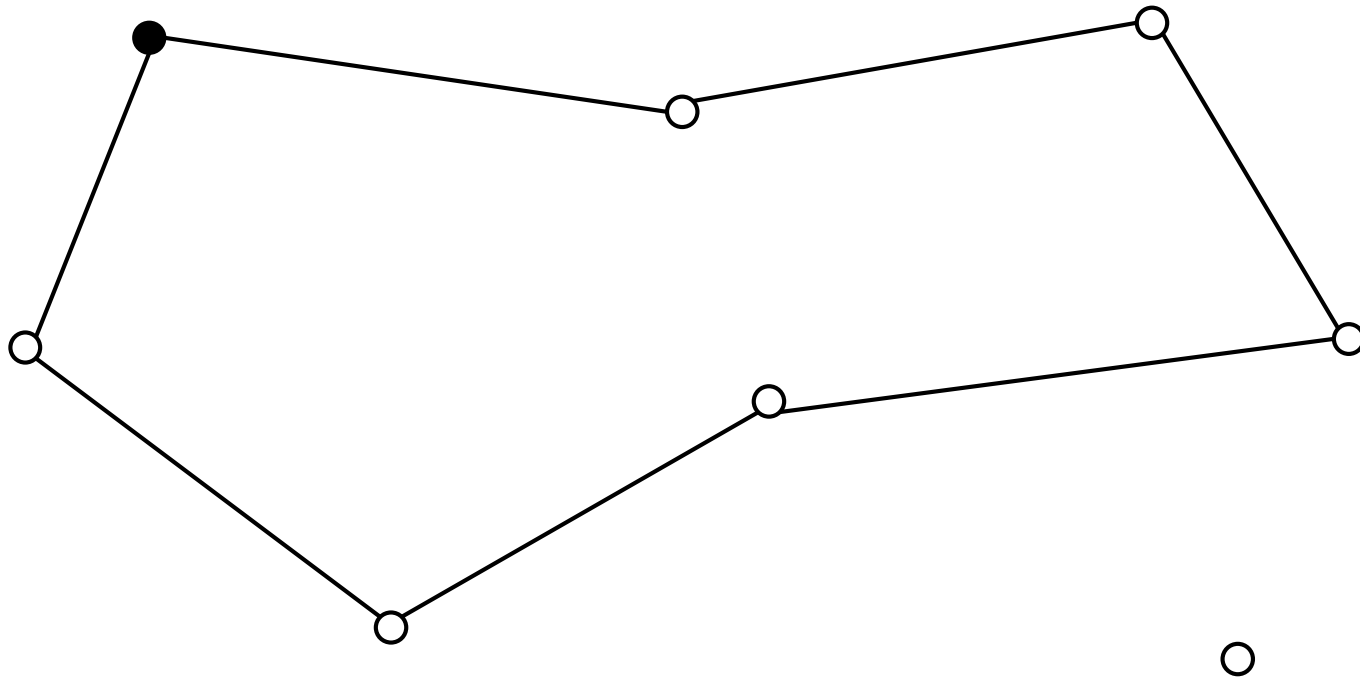
Greedy cycle construction heuristics for TSP



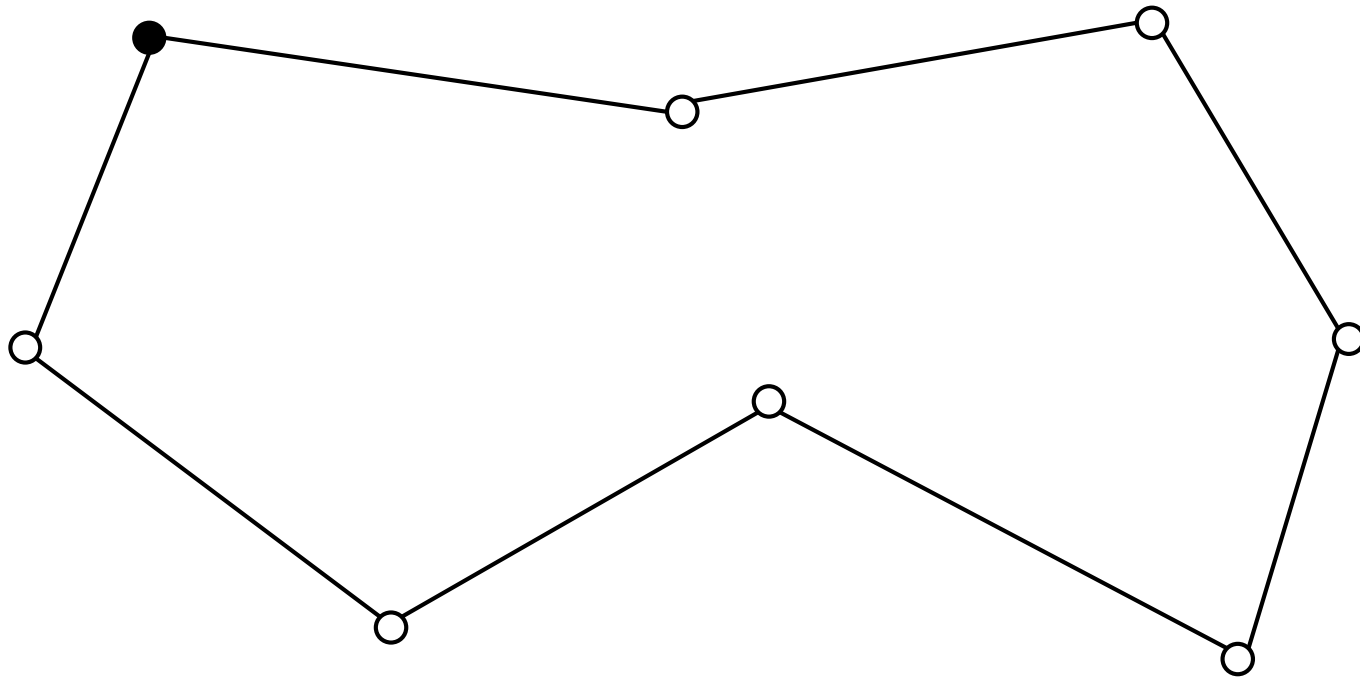
Greedy cycle construction heuristics for TSP



Greedy cycle construction heuristics for TSP



Greedy cycle construction heuristics for TSP



Clarke-Wright heuristic for VRP

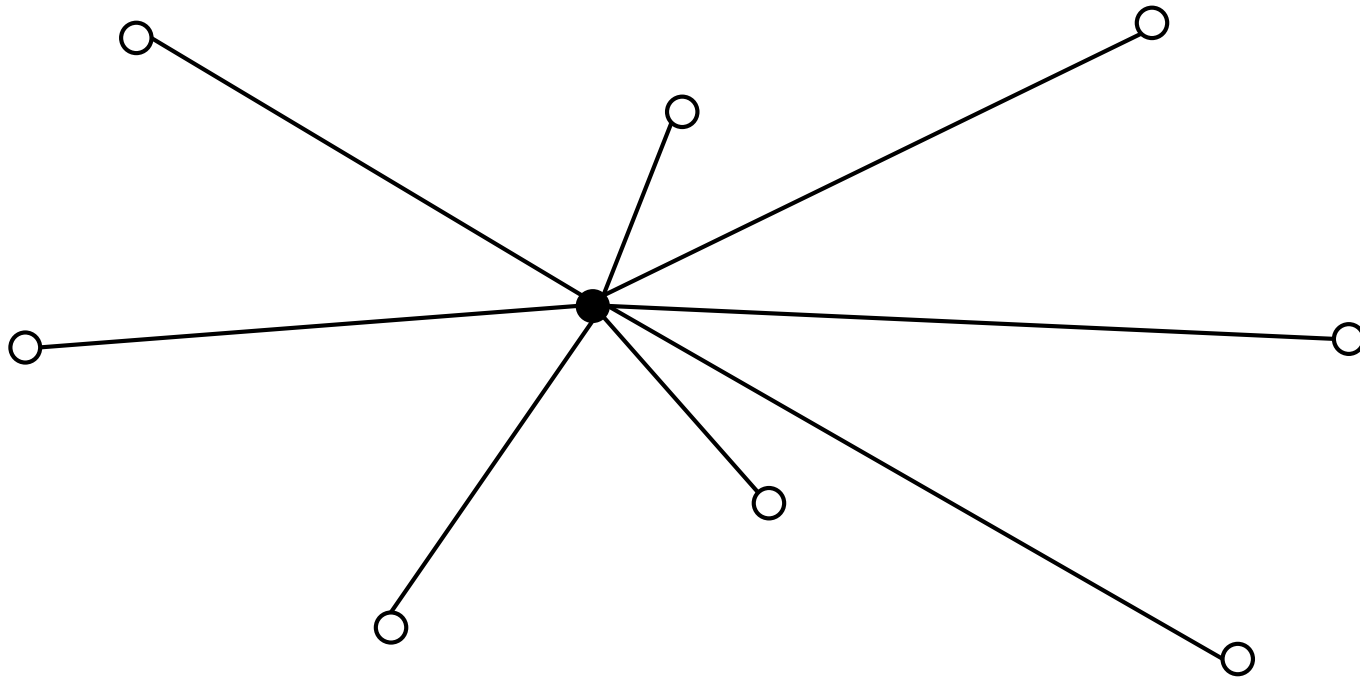
Connect each vertex to the base to create a separate route to each vertex

repeat

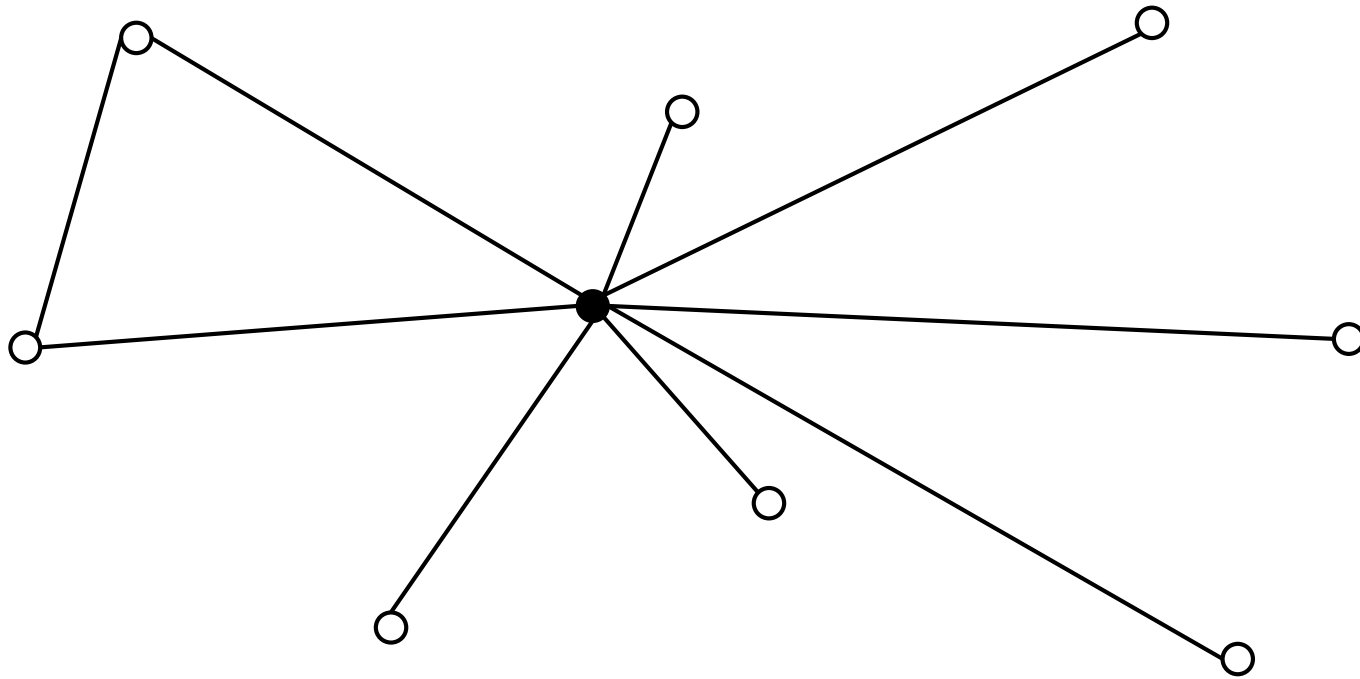
Select and join into one two routes that give the best improvement in the value of the objective function and which joining is feasible

until it is not possible to join any routes without violating constraints

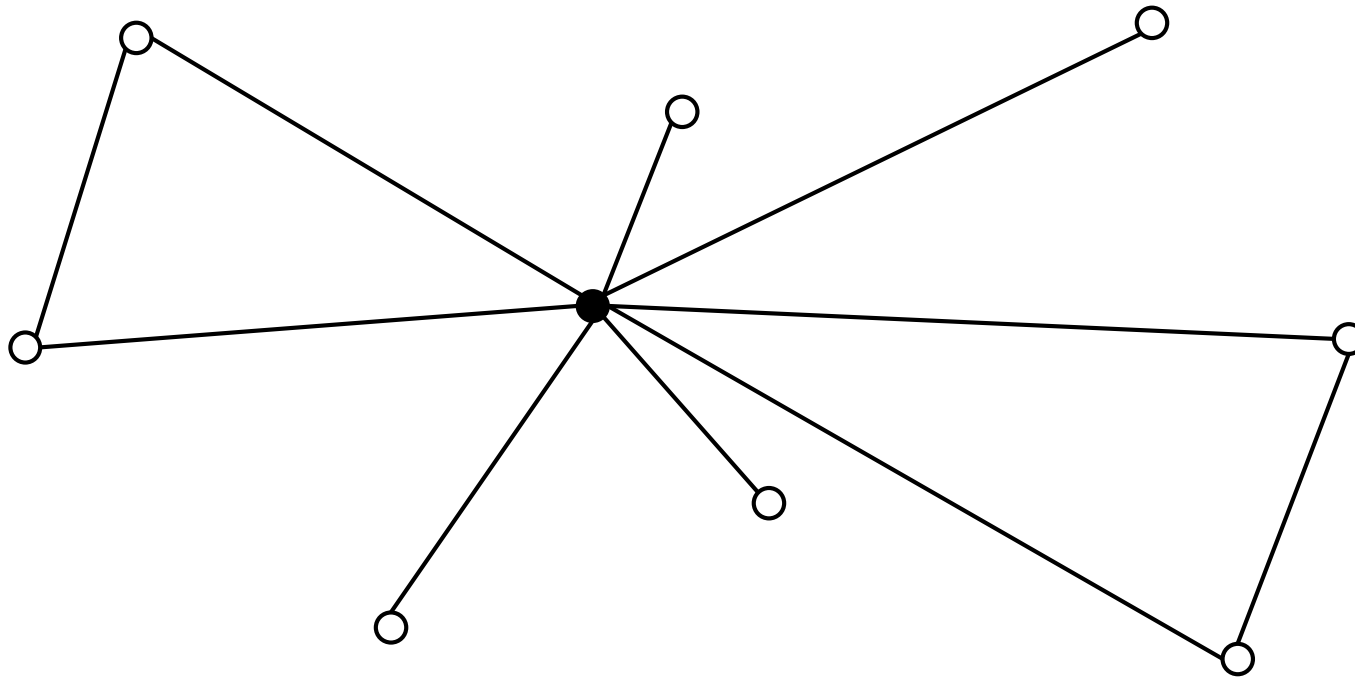
Clarke-Wright heuristic for VRP



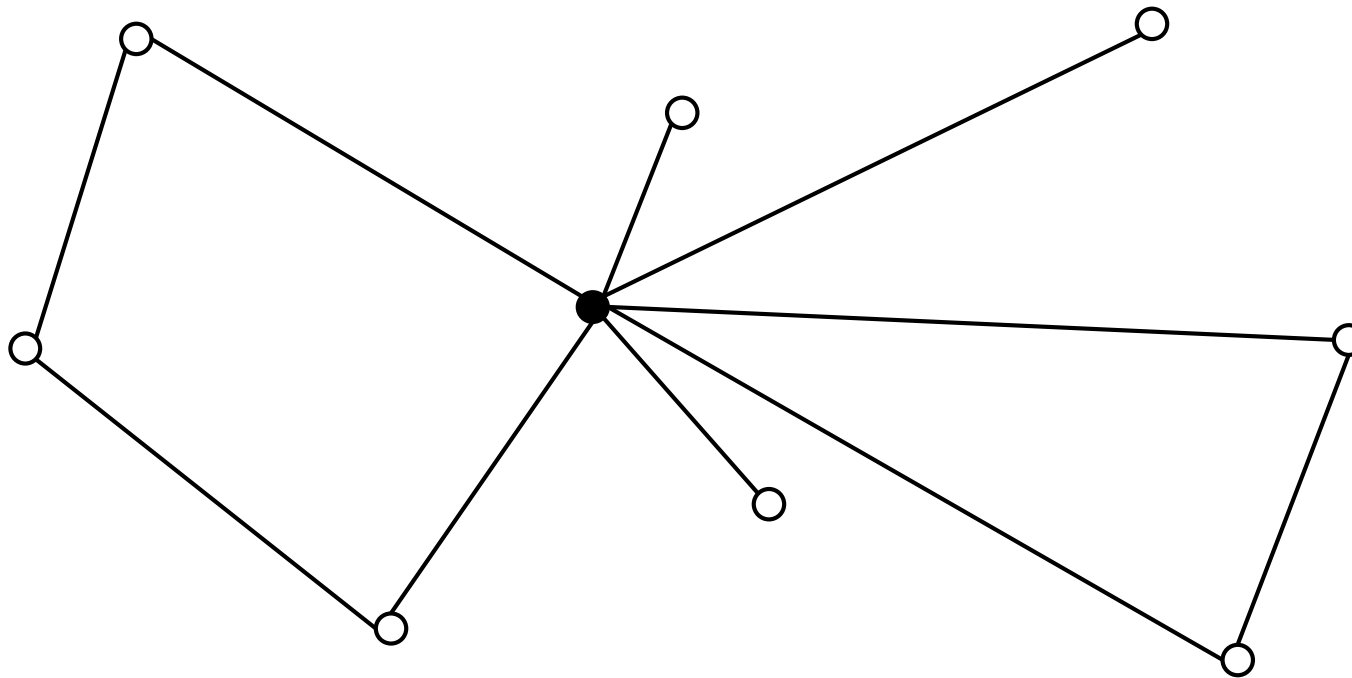
Clarke-Wright heuristic for VRP



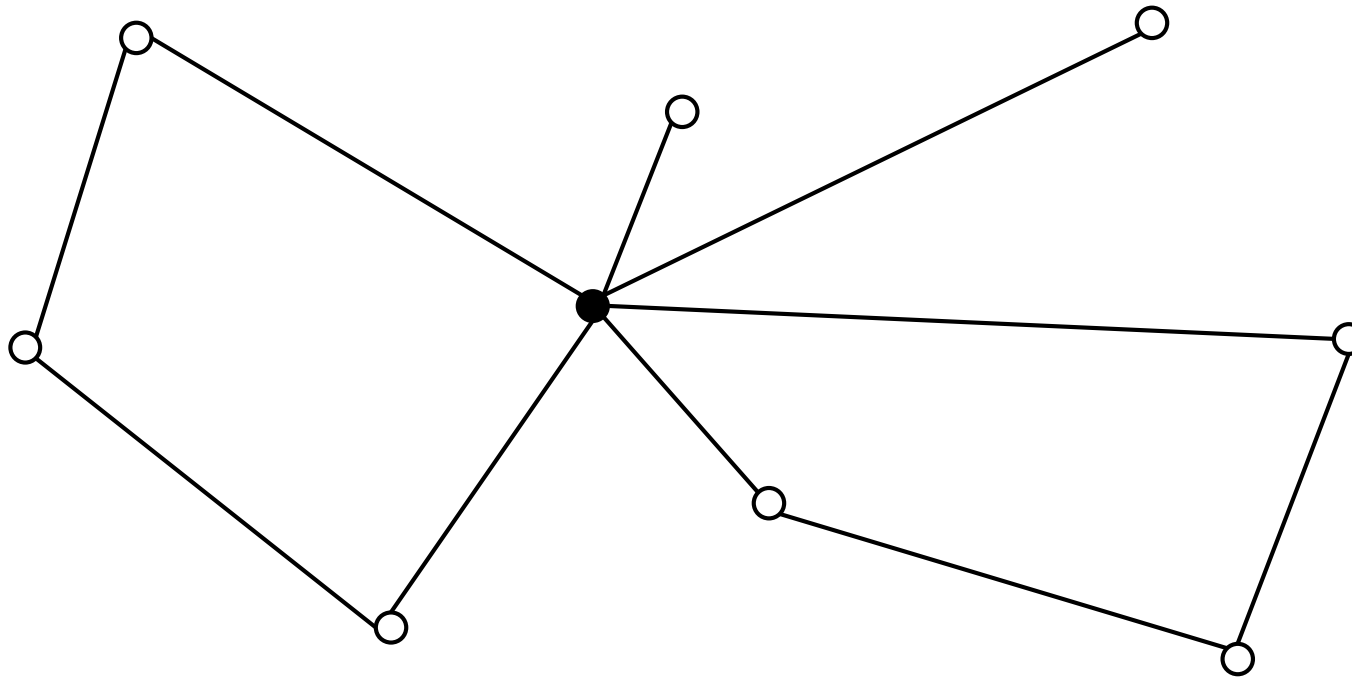
Clarke-Wright heuristic for VRP



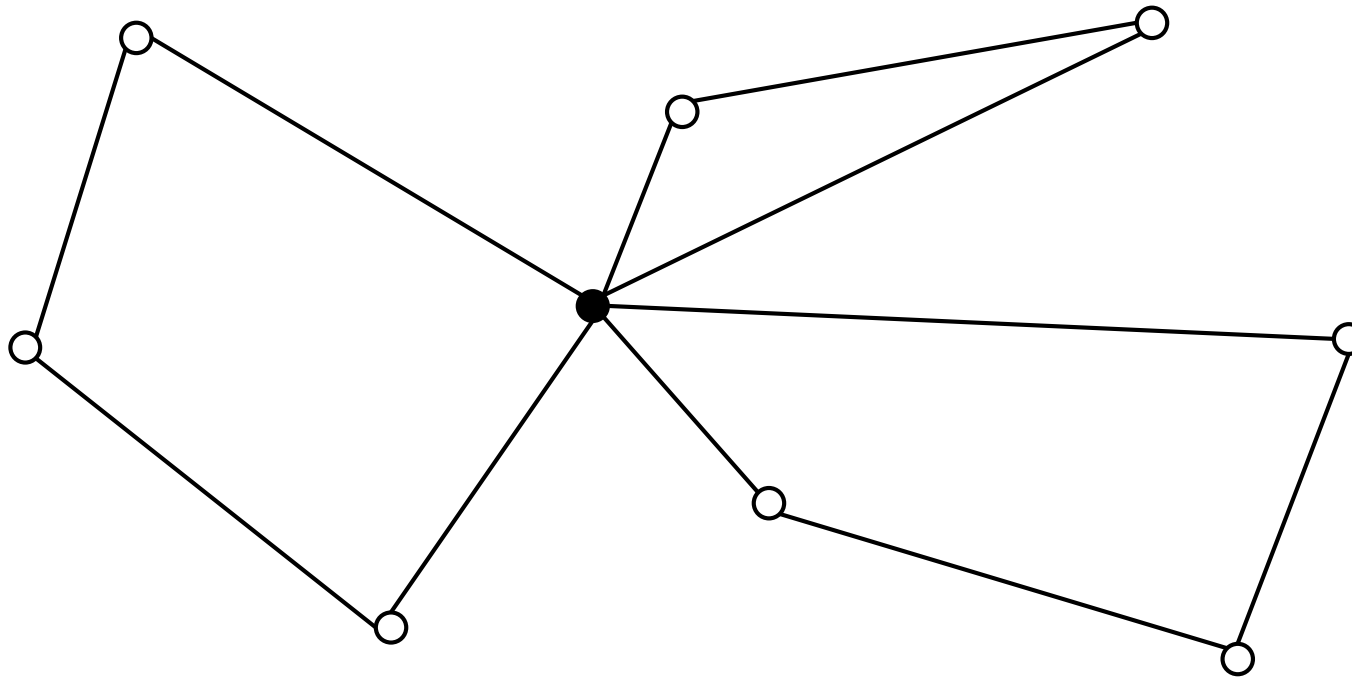
Clarke-Wright heuristic for VRP



Clarke-Wright heuristic for VRP



Clarke-Wright heuristic for VRP



Optimality of greedy construction heuristics

- They are optimal (i.e. they guarantee finding global optimum) for problems that are weighted matroids with non-negative weights where the weight of the sum of the elements is the sum of their weights (e.g. the minimum spanning tree).
 - Matroid is a pair (E, L) where E is a finite base set and L is a set of independent subsets of E i.e. such that
 - The empty set is independent
 - Each subset of an independent set is independent set $A' \subset A \in L$ then $A' \in L$
 - It can be shown that all maximum independent sets have the same number of elements
 - Eg. for the spanning tree E is a set of edges L is a set of all trees

$(1 - 1/e)$ -Approximation of greedy algorithms

- They give some approximation guarantee if the objective function is submodular and monotonic
- Submodular – adding to a superset cannot be better than adding to a subset
 - $A \subseteq B \Rightarrow f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$
- Eg. MaxCover problem – select K subsets maximizing the total number (or weight) of the covered elements
- Hypervolume subset selection problem – select a subset of Pareto-optimal solutions (representation) maximizing the hypervolume quality measure

Practical behavior of greedy heuristics for other problems

- The above assumptions (being a weighted matroid submodularity and monotonicity) can be met „almost/approximately"
- As a result they may generate solutions much better than random solutions

Lazy greedy approach for submodular and monotonic functions

- In some cases calculation of the change of the objective function may be time consuming
- Submodular and monotonic functions:
 - $A \subseteq B \Rightarrow f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$
- Means that changes to the objective function calculated in a previous iteration may be used as upper bounds for changes in any subsequent iteration

Lazy greedy approach for submodular and monotonic functions

Create an incomplete initial solution e.g. $\mathbf{x} := \emptyset$

repeat

if first iteration

 calculate Δf_e for each element e

 use Δf_e as an upper bound Δf_{eUB} for subsequent iterations

 add to \mathbf{x} the currently best element not yet included in the solution

else

 consider elements e not yet included in the solution in decreasing order of Δf_{eUB}

if $\Delta f_{eUB} < \Delta f_{e^*}$

 break the loop

 calculate Δf_e and update $\Delta f_{eUB} = \Delta f_e$

if $\Delta f_e < \Delta f_{e^*}$

 update the best element $e^* = e$

 add to \mathbf{x} e^*

until a full solution is created

Randomization of greedy heuristics

- A disadvantage of greedy heuristics is their determinism
- ... or limited randomization possibility
 - e.g. randomization by selecting a random starting vertex in the nearest neighbor heuristic for TSP

Greedy Randomized Construction used in Greedy Randomized Adaptive Search Procedure (GRASP)

Construct an incomplete initial solution e.g. $\mathbf{x} := \emptyset$

repeat

 create a restricted candidate list (RCL) composed of a number of the best elements not yet included in the solution \mathbf{x}

 add to \mathbf{x} a randomly selected element from RCL

until a full solution is created

How to create RCL

- Quantitative/percentage rule
 - select $p\%$ of the best elements
- Value rule (based on the objective function value)
 - $\Delta f(e)$ – change of the objective function after adding an item e
 - Δf_{\min} i Δf_{\max} – minimum and maximum change of the objective function (we assume maximization i.e. we want to choose the element of the smallest possible $\Delta f(e)$)
 - $\Delta f(e) \in [\Delta f_{\min}, \Delta f_{\min} + \alpha (\Delta f_{\max} - \Delta f_{\min})]$ $\alpha \in (0, 1)$
 - What if $\alpha = 0$?
 - What if $\alpha = 1$?

Another way of randomization

Construct an incomplete initial solution e.g. $\mathbf{x} := \emptyset$

powtarzaj

browse $p\%$ of randomly selected elements (but at least 1) not yet included in the solution and choose the best of them

add the selected element to \mathbf{x}

until a full solution is created

- What if $p = 0\%$?
- What if $p = 100\%$?

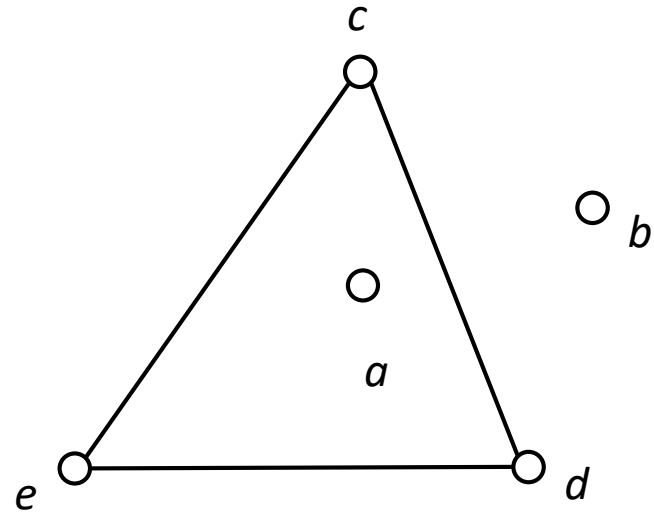
Greedy regret heuristics

- Suppose you can insert an element into the solution in different locations (e.g. vertices in different locations of the TSP route)
- E.g. two elements a and b :
- Insertion costs of a :
 - 1 2 3 4 5
- Insertion costs of b :
 - 5 9 10 12 15
- According to the greedy rule we would choose a ($1 < 5$)
- Then however we can block the possibility of inserting b at cost 5 and we will only have a location with a cost of 9

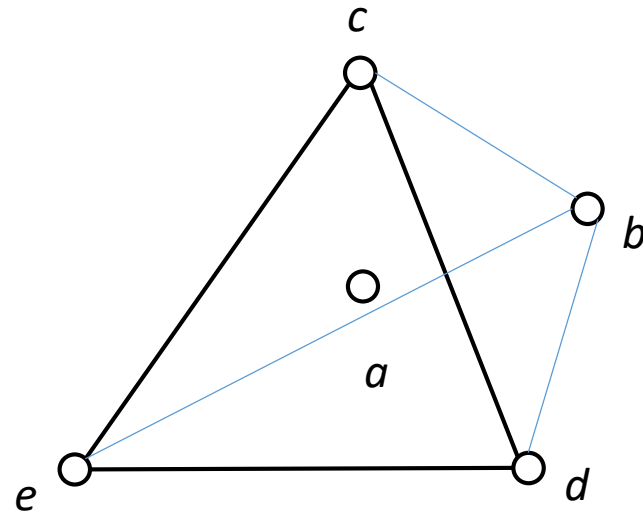
Greedy regret heuristics

- k -regret is the sum of the differences between the best and $k-1$ subsequent insertion options
- 2-regret the difference between the first and second options
- We choose the element with the greatest regret and insert it at the best location
- We can also weigh regret with a greedy rule (cost of the first option)

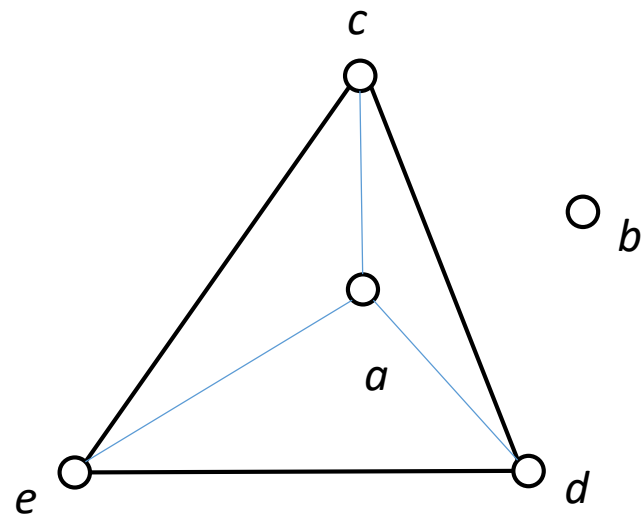
Greedy cycle heuristic



Insertion options of a



Insertion options of b

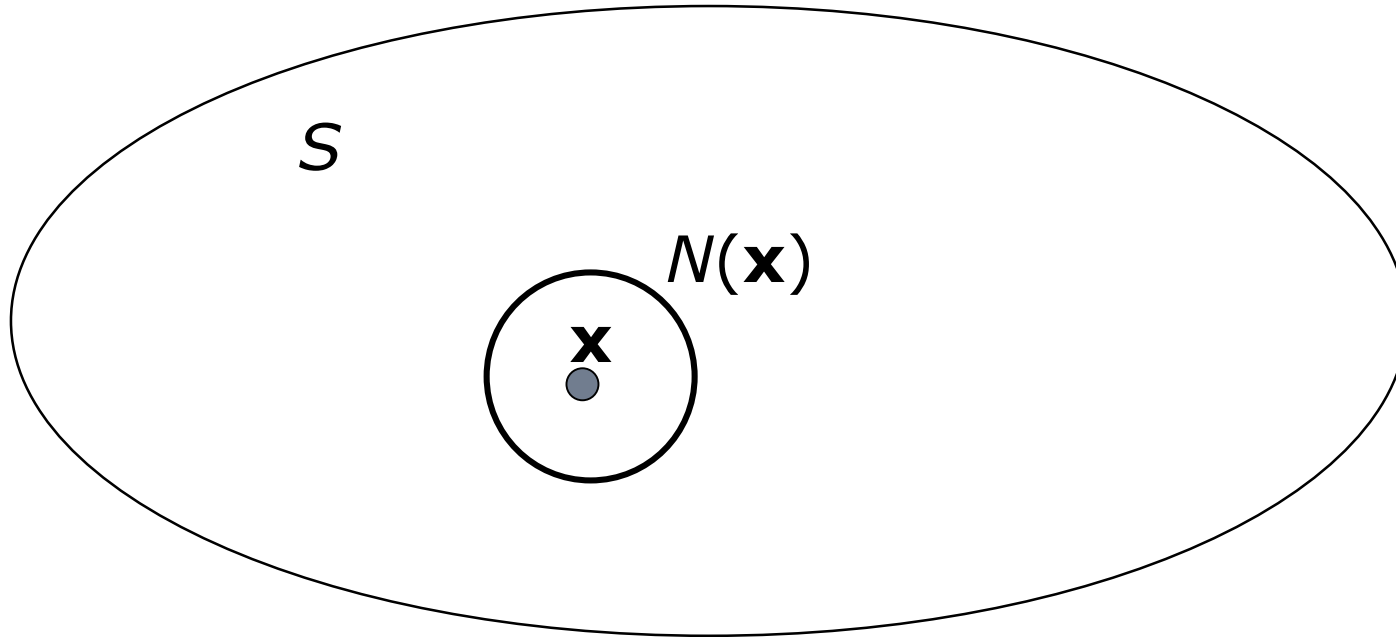


Beam search

- Instead of inserting one element we can evaluate the insertion of several forward (but we only insert the first one)
- This means partially expanding the exhaustive search tree
- In Beam search we consider first adding a certain number of the best elements then a few more best ones (after adding the previous one) etc.
- E.g. Beam search based on NN for TSP
 - We consider a few closest vertices to the last one added for each of them the few closest to this vertex etc.
 - Can be combined with a mechanism of branch-and-bound – reject paths that can not be better than the best already found

Local search

- The idea of the neighborhood – the topology of (feasible) solutions space



Distance-based definition of neighborhood

- $\mathbf{x} \in S$
- set $N(\mathbf{x}) \subset S$ of solutions that are "close" to the solution x
- E.g. distance function

$$d: S \times S \rightarrow \mathbf{R}$$

- neighborhood

$$N(\mathbf{x}) = \{\mathbf{y} \in S : d(\mathbf{x} \ \mathbf{y}) \leq \varepsilon\}$$

- Each solution $\mathbf{y} \in N(\mathbf{x})$ is called a neighbor solution or simply a neighbor of \mathbf{x}

Moves-based definition of neighborhood

- $N(\mathbf{x})$ can be obtained from \mathbf{x} performing one move (modification transformation) m from a certain set of possible moves $M(\mathbf{x})$
- Move m is a transformation that applied to the solution \mathbf{x} gives solution \mathbf{y}
- The neighborhood can therefore be defined as follows:

$$N(\mathbf{x}) = \{\mathbf{y} \in S : \exists m \in M(\mathbf{x}) \mid \mathbf{y} = m(\mathbf{x})\}$$

- This definition is equivalent to the previous one if the distance between \mathbf{x} and \mathbf{y} is defined as the number of moves need to pass from \mathbf{x} to \mathbf{y} and $\varepsilon = 1$

Desirable features of the neighborhood

- Usually we assume that $\mathbf{y} \in N(\mathbf{x}) \Leftrightarrow \mathbf{x} \in N(\mathbf{y})$
 - For a transformation m this means that it is reversible
- Size limitations
 - $N(\mathbf{x})$ must contain at least one (in practice more) solutions different from \mathbf{x}
 - The size of $N(\mathbf{x})$ should be much smaller than the size of the space of all feasible solutions
 - Usually the size of $N(\mathbf{x})$ is polynomial relative to the size of the instance size(although there are exceptions)

Desirable features of the neighborhood

- Similarity of neighbors
 - Performing transformation m should be simple (simpler than constructing a solution from scratch) taking into account:
 - Modification of data structures
 - Re-calculation of the objective function value
 - Checking constraints
- Equality
 - From any solution it should be able to pass to any other solution
 - May not be easy for constrained problems

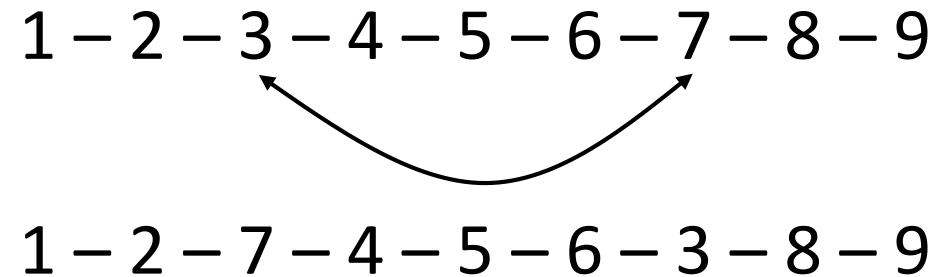
Typical neighborhood operators

- Changing the value of a single element e.g. changing the binary value to the opposite value (e.g. changing the selection of an item in the knapsack problem changing the assignment of an item to a task in the assignment problem)
- Moving (shifting) one element from a set to another set (e.g. between VRP routes)
- Exchange of two elements between two sets (e.g. between VRP routes)
- Moving an element in a sequence (e.g. TSP scheduling)
- Exchange of the position of two elements in a sequence (e.g. TSP scheduling)
- Reversing the order of a part of a sequence (e.g. TSP scheduling)
- These operators can also concern a larger (albeit rather small) number of elements

Defining a neighborhood

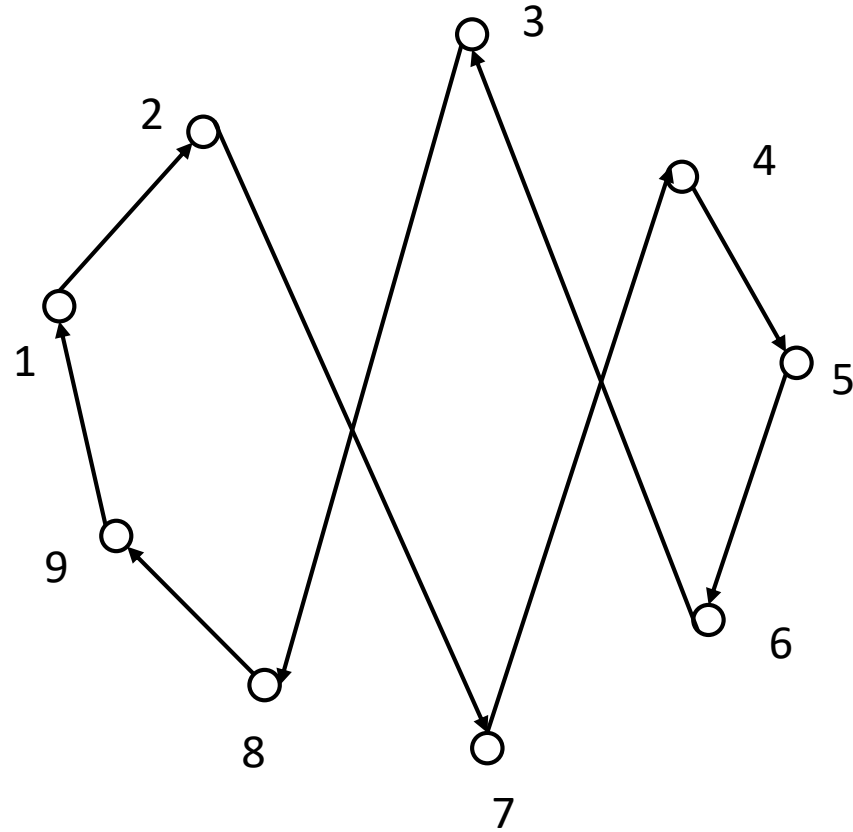
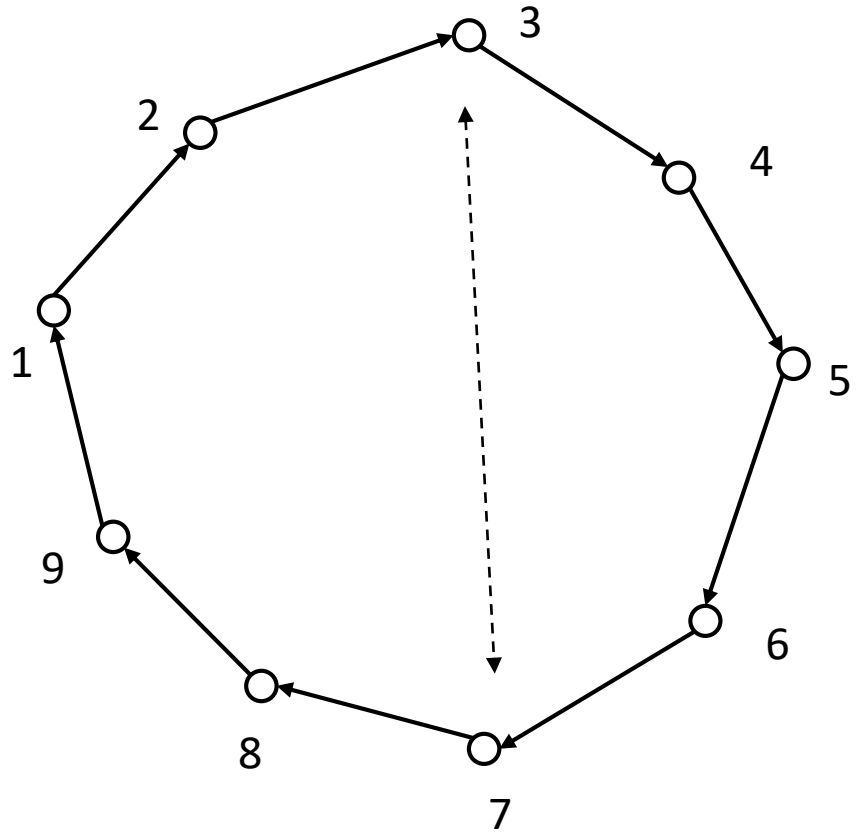
- Neighborhood is not an objective feature of the problem we often have many options to choose (e.g. different operators)
- On the other hand only some neighborhoods meet the above characteristics

Example: Exchange of two vertices in TSP



- Neighborhood size:
 - $|N(x)| = n(n-1)/2 = O(n^2)$
 - Where n is the number of vertices

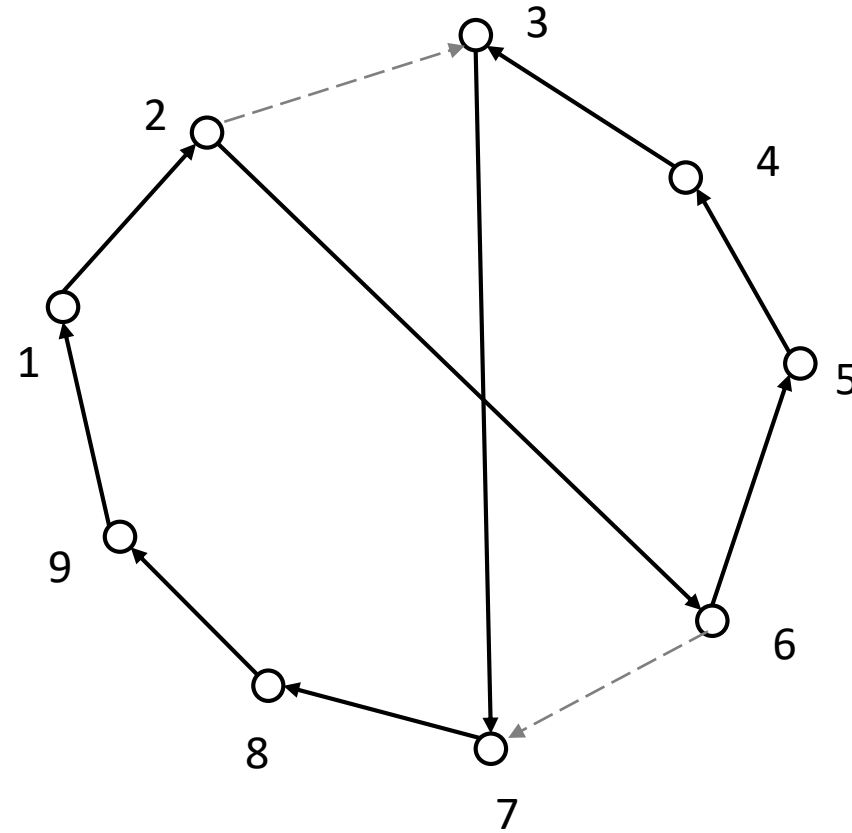
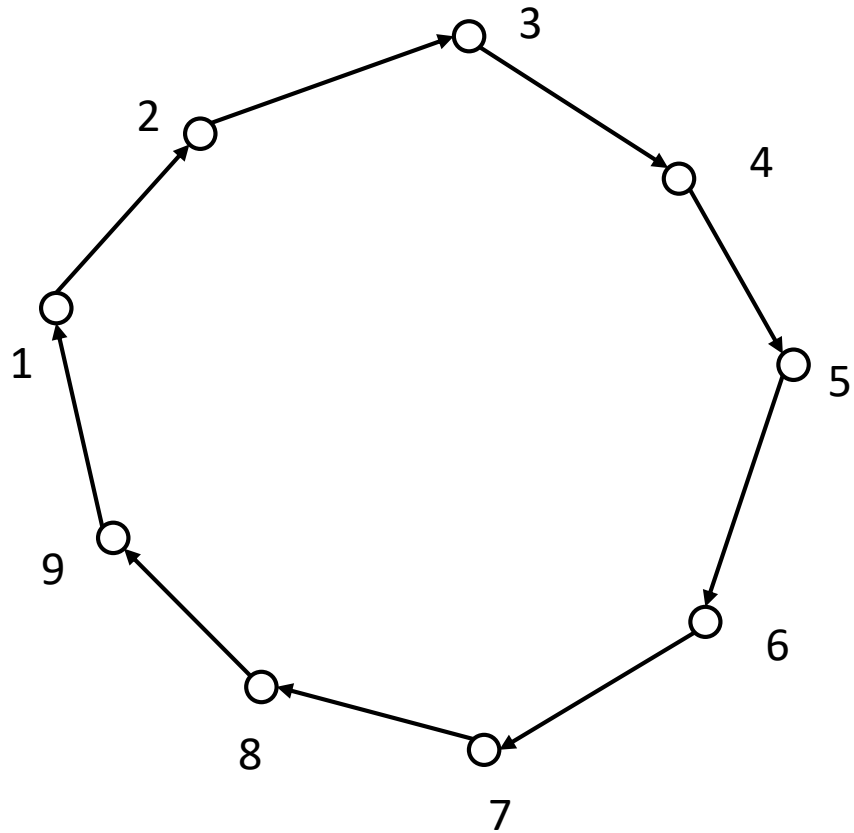
Example: Exchange of two vertices in TSP



Exchange of K vertices in TSP

- Another point of view: we remove K vertices from the the solution and insert them into the freed places in a different way
 - We have $K! - 1$ other than current options of insertion – number of permutations other than the current one
 - Neighborhood size $O\left(\binom{n}{K} K!\right) = O\frac{n!}{(n-K)!} = O(n^K)$

Exchange of two edges in TSP




Exchange of two edges in TSP

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

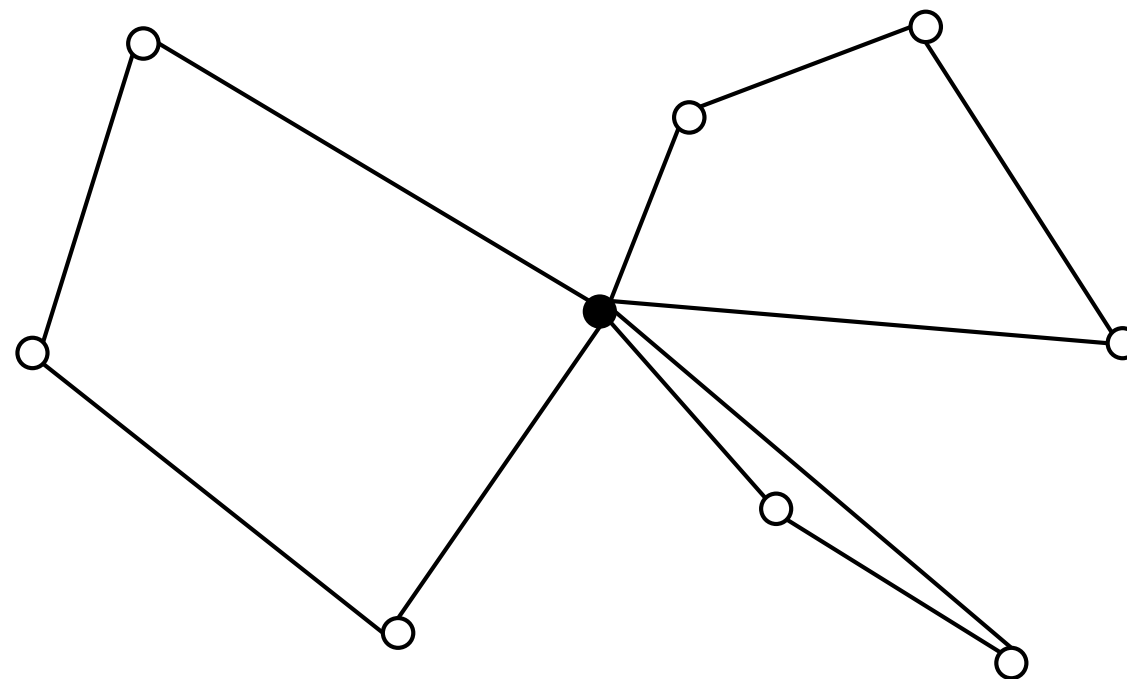
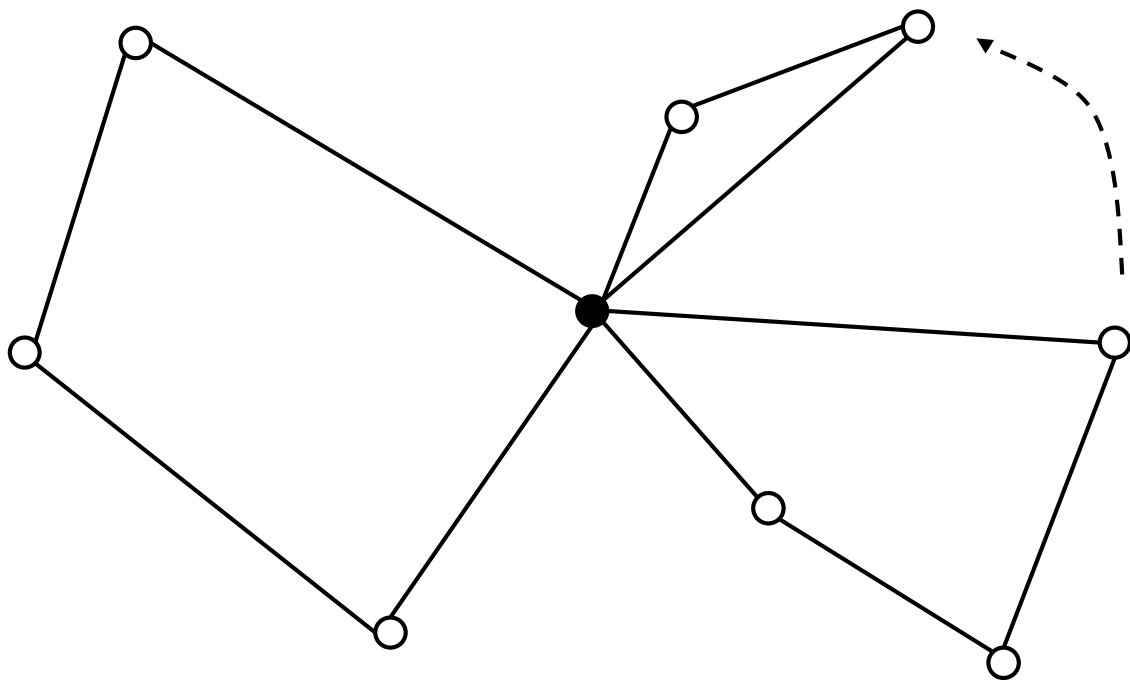
→

1 - 2 - 6 - 5 - 4 - 3 - 7 - 8 - 9

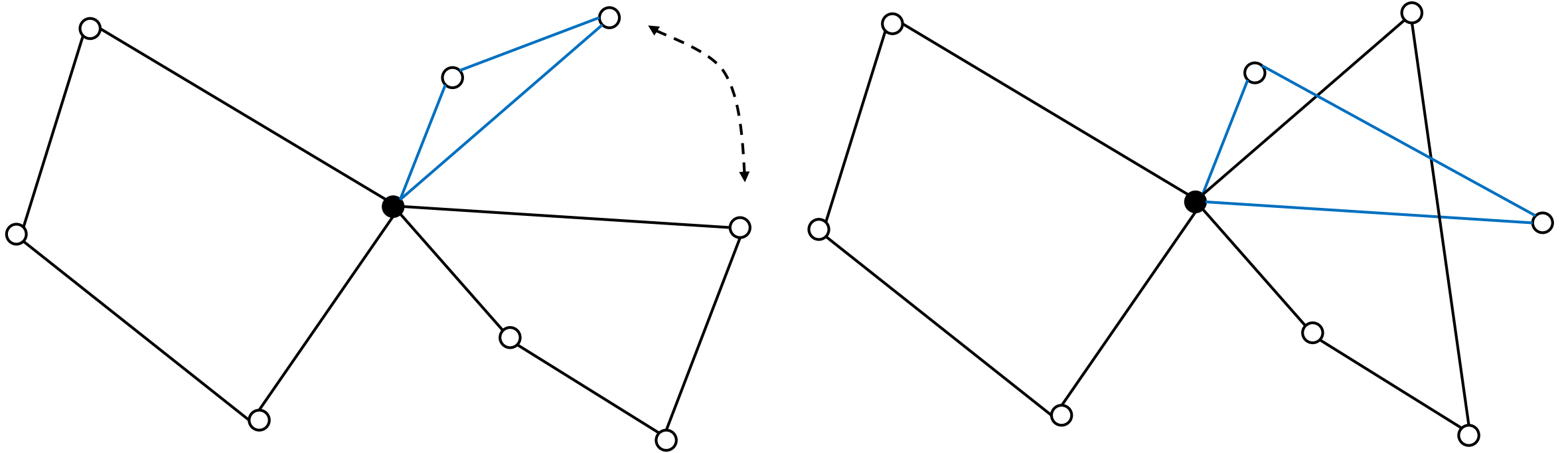


- Neighborhood size:
 - $|N(x)| = n(n-3)/2 = O(n^2)$
- One can also exchange K vertices

Shift of a vertex in VRP



Exchange of two vertices in VRP



VRP neighborhood sizes

- Vertex shift
 - $|N(\mathbf{x})| = n (T - 1) = O(n T)$ where T is the number of routes
- Vertex exchange
 - $|N(\mathbf{x})| \approx n ((n - n / T)) / 2 = O(n^2)$

General idea of local search

- Create an initial solution
 - Random or using a (randomized) heuristics
- Improve the solution by moving to neighbor solutions (making moves) accepting only solutions (moves) that improve the current solution as long as possible (i.e. there are neighbor solutions/moves that bring improvement)
- Also called Hill-climber

The concept of local optimum

\mathbf{x}_{min} is the local minimum if
 $f(\mathbf{x}_{min}) \leq f(\mathbf{y})$ for all $\mathbf{y} \in N(\mathbf{x}_{min})$

\mathbf{x}_{max} is the local maximum if
 $f(\mathbf{x}_{max}) \geq f(\mathbf{y})$ for all $\mathbf{y} \in N(\mathbf{x}_{max})$

Local search in steepest and greedy versions (maximization assumed)

Greedy version

Generate initial solution \mathbf{x}

repeat

for each $\mathbf{y} \in N(\mathbf{x})$ in a random order

if $f(\mathbf{y}) > f(\mathbf{x})$ **then**

$\mathbf{x} := \mathbf{y}$

until no better solution was found after
checking the whole $N(\mathbf{x})$

Steepest version

Generate initial solution \mathbf{x}

repeat

 find the best solution $\mathbf{y} \in N(\mathbf{x})$

if $f(\mathbf{y}) > f(\mathbf{x})$ **then**

$\mathbf{x} := \mathbf{y}$

until no better solution was found after
checking the whole $N(\mathbf{x})$

Local search in the greedy version

- The first improving solution is accepted
- If the neighborhood (moves) are evaluated in a certain fixed order this may lead to a bias of the algorithm (e.g. preferring the exchange of certain vertices)
- The ideal approach is to evaluate the neighborhood (moves) in random order
- An acceptable approach is a reasonable randomization that eliminates most of the bias e.g. starting a loop over possible moves from a random index and drawing at random the direction of the loop

Greedy vs. steepest version

- The greedy version usually is faster but may give worse solutions than the steepest
- The comparison is not unambiguous because during a single run of the steepest version one can often run the greedy version several times and choose the best solution
- Since greedy LS accepts the first improving solution the steps are generally smaller in terms of objective function change. At the same time the ranges between the starting and final (locally optimal) values are very similar to the steepest LS. Thus greedy LS performs on average more steps before achieving a local optimum and changes the starting solution more.

Time efficiency of local search

- In most of the more advanced algorithms that use local search LS will consume most of the running time
- In LS itself the critical operation (bottleneck) is the evaluation of neighbor solutions. The number of solutions evaluated is much greater than the number of solutions accepted
 - In the steepest version one accepted solution after evaluating the entire neighborhood
 - In the greedy version when the current solution is relatively good most of the moves also result in a deterioration
- Therefore the evaluation of solutions should be refined as much as possible
 - other parts of the algorithm often do not need to be optimized for efficiency

Basic rule of code optimization ;-)



Direct (trivial) implementation of local search

- Neighbor solutions are explicitly constructed before evaluation. The objective function is calculated from the scratch
 - e.g. by summing edge lengths in TSP

Calculation and use of the change (delta) of the objective function

- To evaluate a neighbor solution given by move m it is enough to calculate the delta of f : $\Delta f_m(\mathbf{x})$
- For the maximized function the solution $m(\mathbf{x}) \in N(\mathbf{x})$ is better than the current one (\mathbf{x}) if:
 - $\Delta f_m(\mathbf{x}) > 0$
- move $m1$ is better than $m2$ if:
 - $\Delta f_{m1}(\mathbf{x}) > \Delta f_{m2}(\mathbf{x})$
- Preferably move evaluation should be constant time

Local search with the use of deltas

Steepest version

Generate an initial solution \mathbf{x}

repeat

find the best move $m \in M(\mathbf{x})$

if $f(m(\mathbf{x})) > f(\mathbf{x})$ **then**

$\mathbf{x} := m(\mathbf{x})$

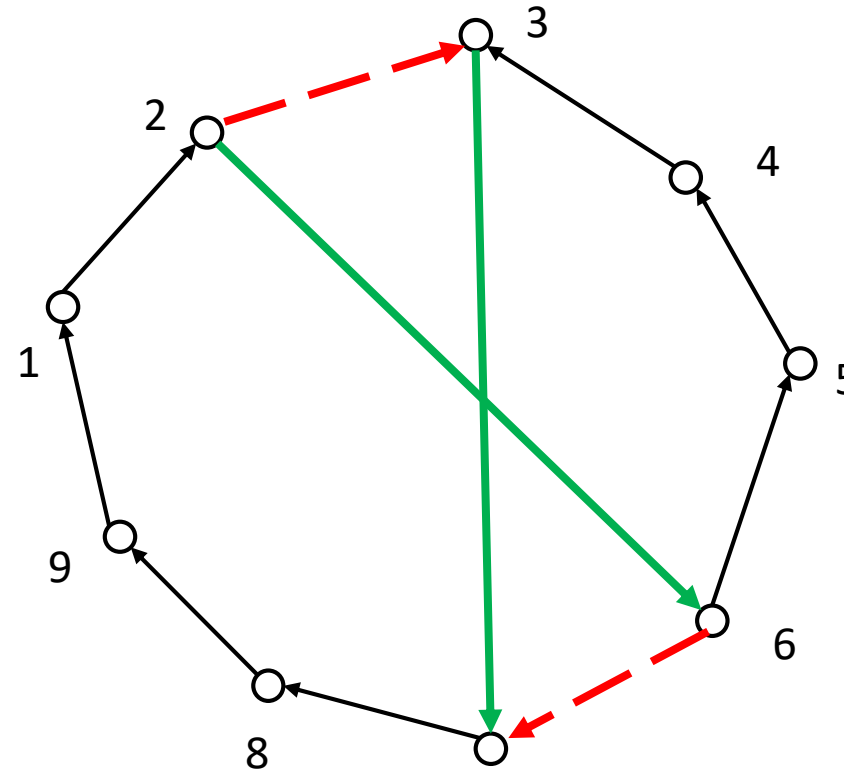
until no better solution was found after
checking the whole $M(\mathbf{x})$

Moves (changes to f) are
evaluated not solutions!
Neighbor solutions do not
have to be explicitly
constructed!

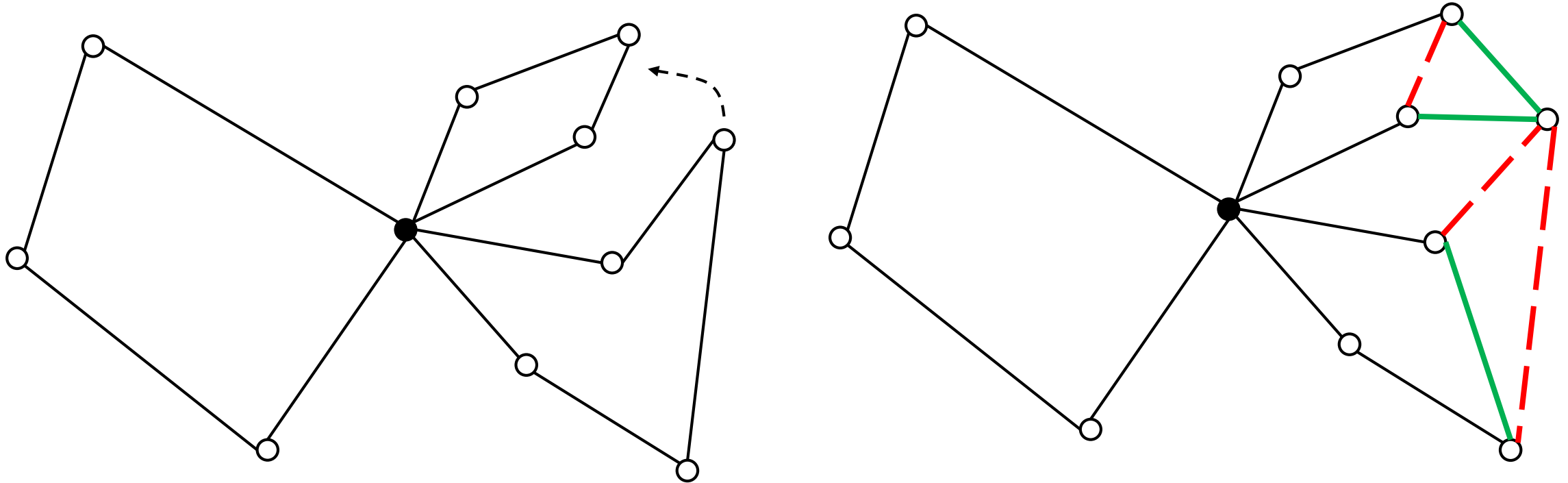
Only here is the current
solution modified

Delta in the Traveling Salesperson Problem (TSP) – 2-edge exchange

- To calculate the delta simply subtract the lengths of the removed edges and add the lengths of the added edges – 4 arithmetic operations
- Calculating f from the scratch requires n operations



Delta in the VRP – vertex shift



Adding the length of three edges and subtracting the length of three edges – 6 operations

Delta in the VRP – vertex shift

- Number of moves:
 $O(n^2) \approx n^2/2$
- Calculation of f from the scratch (without delta)
 - One need to add up the lengths $O(n)$ edges
 - The complexity of one iteration $O(n^2) O(n) = O(n^3) \approx n^3/2$
- Calculation of a delta
 - 6 operations
 - The complexity of one iteration $\approx 6 n^2/2 = 3 n^2$
- Speed-up
 - $\approx n/6$

The use move evaluations (deltas) from previous iterations of local search

- For $\mathbf{y} \in N(\mathbf{x})$ usually $N(\mathbf{x}) \cap N(\mathbf{y}) = \emptyset$ or $|N(\mathbf{x}) \cap N(\mathbf{y})| \ll |N(\mathbf{x})|$
but
- $M(\mathbf{x}) \cap M(\mathbf{y})$ can be numerous i.e. many moves remain the same after performing the previous move
 - So although the neighboring solutions are different many of the moves that can be applied and their evaluations (deltas) remain the same – they do not need to be re-evaluated.

Local search in steepest version

Generate an initial solution \mathbf{x}

repeat

 find the best move $m \in M(\mathbf{x})$

if $f(m(\mathbf{x})) > f(\mathbf{x})$ **then**

$\mathbf{x} := m(\mathbf{x})$

until no better solution was found after checking the whole $M(\mathbf{x})$

The use of deltas for VRP – vertex shift

- Number of vertex shifts
 - $\approx n \times (T - 1) \times n / T \approx n^2$ where T is the number of routes



- Cost of evaluation of a new solution (move)
 - $\approx n$
- Total cost of one iteration without delta
 - $\approx n^3$
- Total cost of one iteration with delta
 - $\approx 6n^2$

The use move evaluations (deltas) from previous iterations of local search in VRP

- Let's first assume that we recalculate all moves concerning modified routes (not just modified positions)
 - This may be required in some cases e.g. when time windows affect the entire route

The use of move evaluations (deltas) from previous iterations of local search – vertex shift

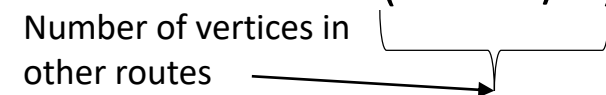
- In the previous iteration only two routes has been changed
- Only moves concerning the changed routes should be recalculated (using the delta)
- Number of vertex shift from these routes

- $\approx 2 n / T \times (T - 1) \times n / T \approx 2 n^2 / T$ where T is the number of routes



- Number of vertex shifts to these routes

- $\approx (n - 2 n / T) 2 \times n / T \approx 2 n^2 / T$



- Total number of moves concerning modified routes
 - $\approx 4 n^2 / T$

The use of move evaluations (deltas) from previous iterations of local search – vertex shift

- Total number of new moves that need to be evaluated
 - $\approx 4n^2 / T$
- The complexity of one iteration with delta
 - $\approx (4n^2 / T) \times 6 = 24n^2 / T$
- Speed-up relative to the base version with delta
 - $\approx (6n^2) / (24n^2 / T) = T / 4$
- Speed-up relative to the base version without delta
 - $\approx n^3 / (24n^2 / T) = n \times T / 24$
- For $n=1000$ $T = 100$ (fully realistic in practice) speed-up relative to the base version without delta is 4166 and relative to the base version with delta is 25 - without changing the results of the algorithm

The use of move evaluations (deltas) from previous iterations of local search in VRP

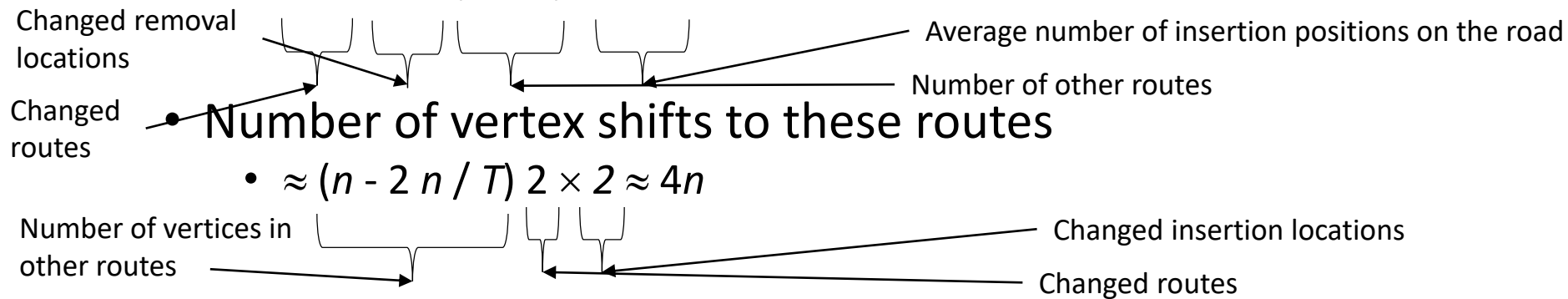
- However it is enough to re-calculate the moves whose surroundings have changed i.e. the neighbor vertices has changed – only 2 per route

The use of move evaluations (deltas) from previous iterations of local search in VRP

- It is enough to re-calculate the moves whose surroundings have changed i.e. the neighbor vertices has changed – only 2 per route

- Number of vertex shifts from these routes

- $\approx 2 \times 2 \times (T - 1) \times n / T \approx 4n$



- $\approx (n - 2n / T) 2 \times 2 \approx 4n$

- Total number of moves that need to be re-evaluated
 - $\approx 8n$

The use move evaluations (deltas) from previous iterations of local search – vertex shift

- Total number of moves that need to be re-evaluated
 - $\approx 8n$
- The complexity of one iteration with a delta
 - $\approx 8n \times 6 = 48n$
- Speed-up relative to the base version with delta
 - $\approx (6n^2) / (48n) = n / 8$
- Speed-up relative to the base version without delta
 - $\approx n^3 / (48n) = n^2 / 48$
- For $n=1000$ $T = 100$ (fully realistic in practice) speed-up relative to the base version without a delta is 20833 and relative to the base version with a delta is 125 - without changing the results of the algorithm

The use move evaluations (deltas) from previous iterations of local search

- Technical approaches
 - Cache of moves from the previous iteration
 - or
 - List of improving moves sorter according to the delta
 - The associated overheads may reduce the effects – effective speed-up will be lower than theoretical (or even none)
- Sometimes we cannot reuse the whole deltas but some intermediate steps in deltas calculation

List of improving moves sorted according to the delta

Initiate LM – a list of moves that bring improvement ordered from the best to the worst

Generate an initial solution \mathbf{x}

repeat

Evaluate all **new** moves and add improving moves to LM

for moves m from LM starting from the best until a applicable move is found

 Check if m is applicable and if not remove it from LM

if move m has been found **then**

$\mathbf{x} := m(\mathbf{x})$ (accept $m(\mathbf{x})$)

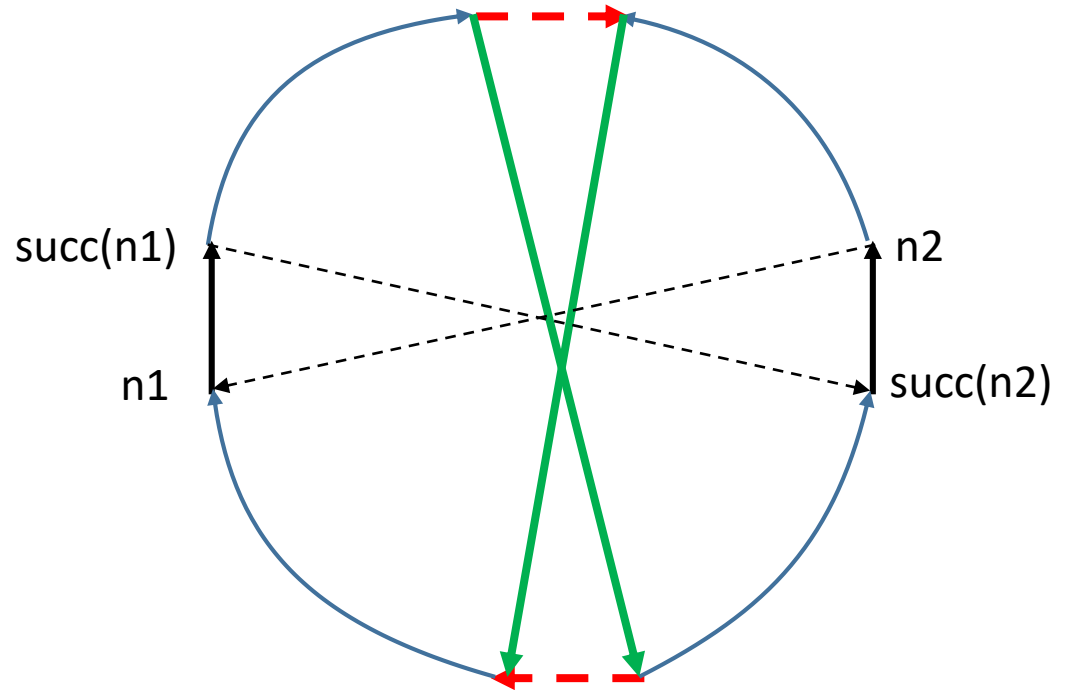
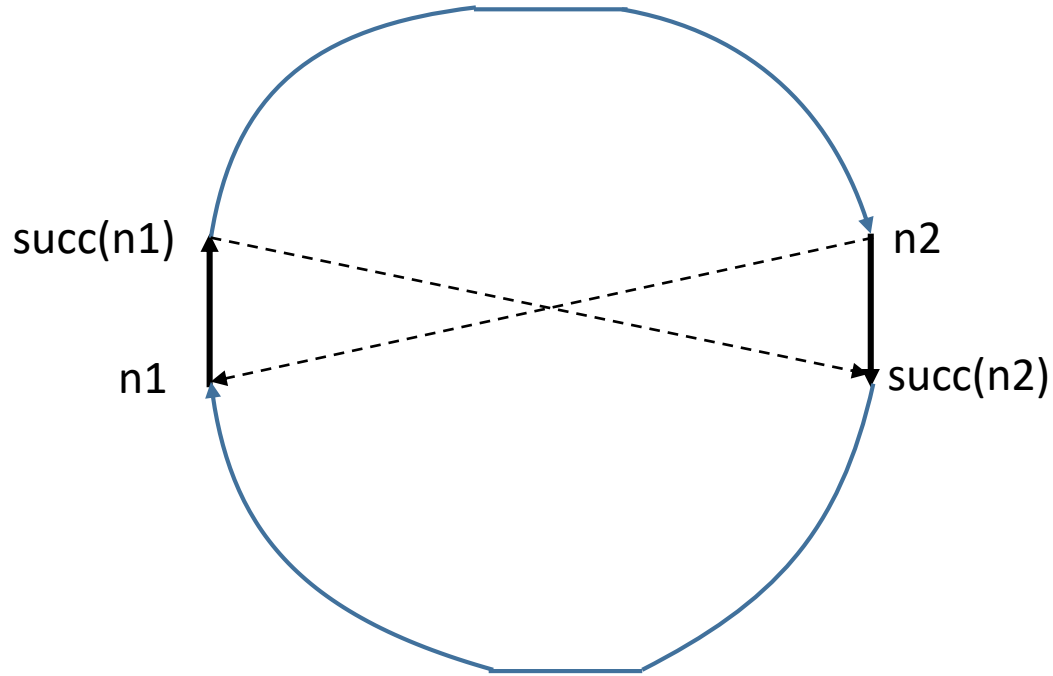
until no move has been found after checking the whole list LM

Applicable moves

- A move may be non-applicable (it cannot be applied) due to changes in the solution caused by other performed moves

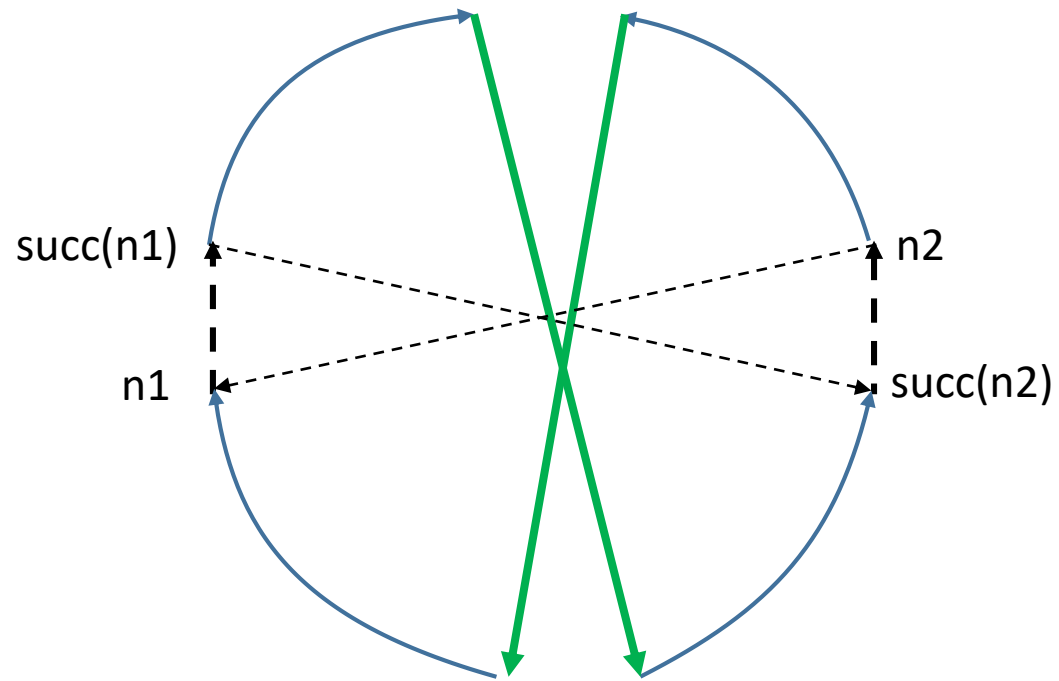
Example for TSP

Evaluate the move associated with the removal of edges $(n1, \text{succ}(n1))$, $(n2, \text{succ}(n2))$ and adding edges $(n1, n2)$, $(\text{succ}(n1), \text{succ}(n2))$

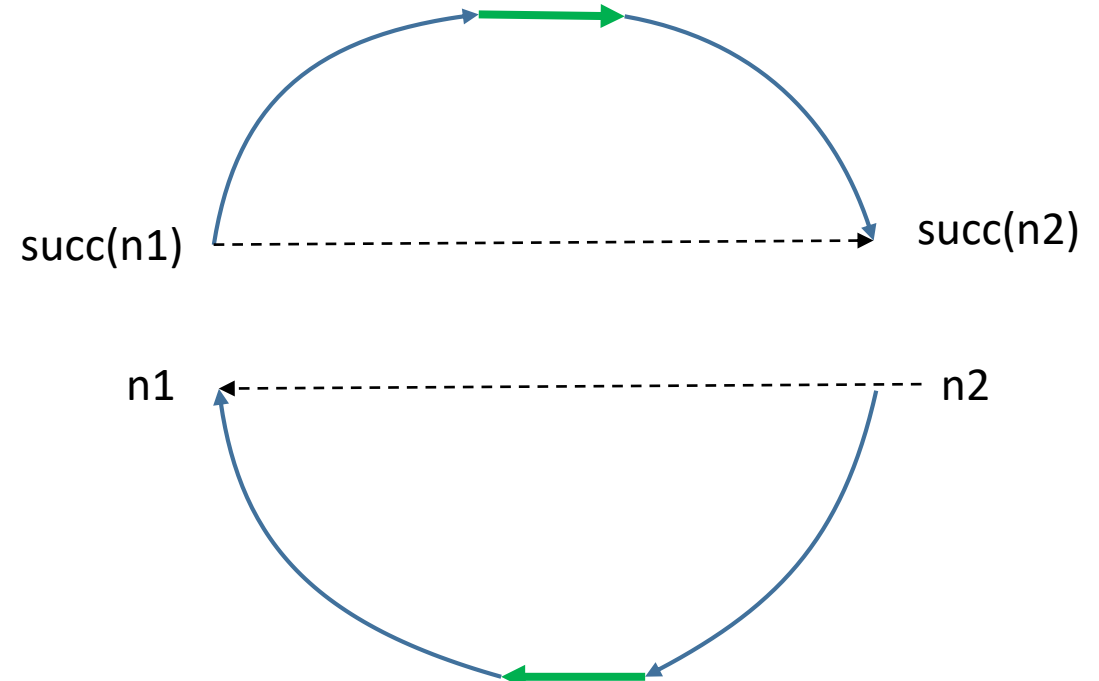


Another move is performed

Example for TSP



After performing another move



The saved move is not correct (applicable)
but it can become correct after another move

Conclusions from the above example

- We must also take into account the relative direction of the edges in the current solution
- Any other move can change this relative direction
- 3 situations (when we browse moves from *LM*):
 - Removed edges (defining the saved move) no longer exist in the current solution (at least one of them)
 - -> remove the move from *LM*
 - Removed edges occur in the current solution in a different relative direction from the saved one – not applicable now but the move can be applied in the future
 - -> leave the move in *LM* but do not apply it browse *LM* further
 - Removed edges appear in the current solution in the same relative direction (also both reversed)
 - -> perform (apply) the move and remove from *LM*
- When evaluating new moves we need to consider also moves with inverted edges

Global memory of deltas

- So far we have assumed remembering current deltas within a single run of LS
- However local search can be repeated many times within a higher level method
- Moves and their deltas can therefore be repeated during various LS runs
- So one can remember all the already calculated deltas in a global memory (which however may be very large)
 - Hashing to improve efficiency
 - Memory management techniques such as deleting the least used deltas

Global memory of deltas

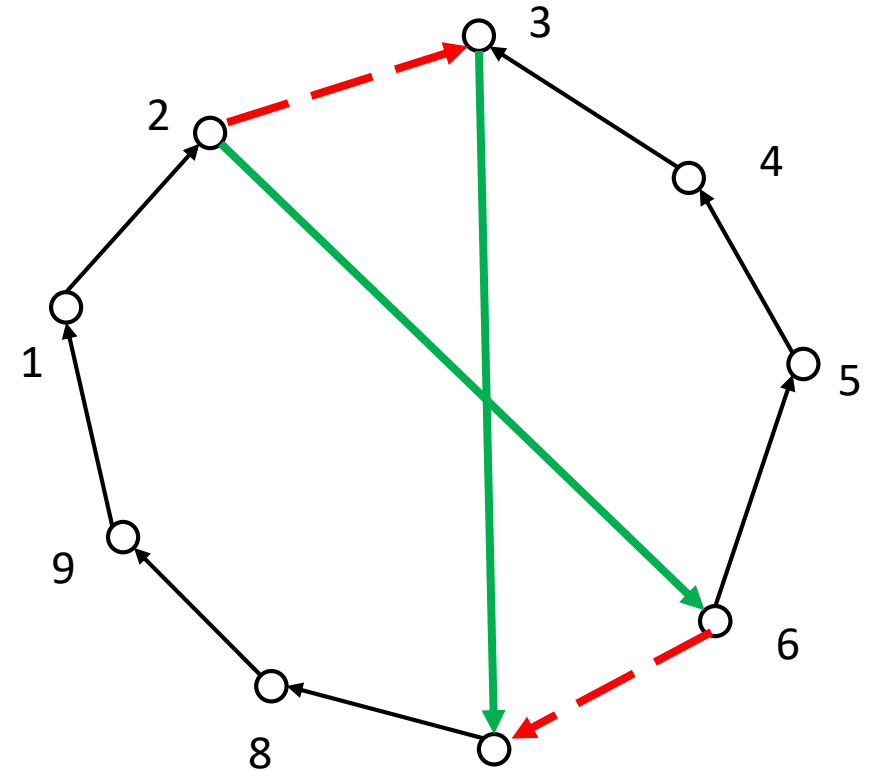
- For example for TSP there are $O(n^4)$ different two edges exchange moves (four edges) and $(n - 1)!$ solutions

Candidate moves - skipping non-promising moves based on heuristic rules

- Ideally one should only evaluate the move that will be accepted (performer)
 - Bringing improvement in the greedy version
 - The best in the steepest version
- In practice in general impossible but one could skip (not evaluate) many non-promising moves on the basis of heuristic rules
- A technique also known as *candidate moves*
- Candidate (promising) moves are evaluated in the neighborhood
- Other moves are omitted or evaluated with a small probability
- In the ideal version we reduce the calculation time without compromising quality
 - In practice if some good moves are not on the candidate list there is a deterioration in the quality of the solutions found.

Example of TSP (similarly VRP)

- For each vertex we create a list of length $n' \ll n$ of the nearest vertices – and the resulting candidate edges
- Candidate moves introduce at least one candidate edge leading to one of the nearest vertices



Standard TSP neighborhood search – two-edge exchange – steepest version

For each vertex n_1 from 0 to $N-1$

For each vertex n_2 from n_1+1 to $N-1$

If edges $n_1\text{-succ}(n_1)$ i $n_2\text{-succ}(n_2)$ are not adjacent

Evaluate if move $(n_1\ n_2)$ brings improvement and is the best found so far

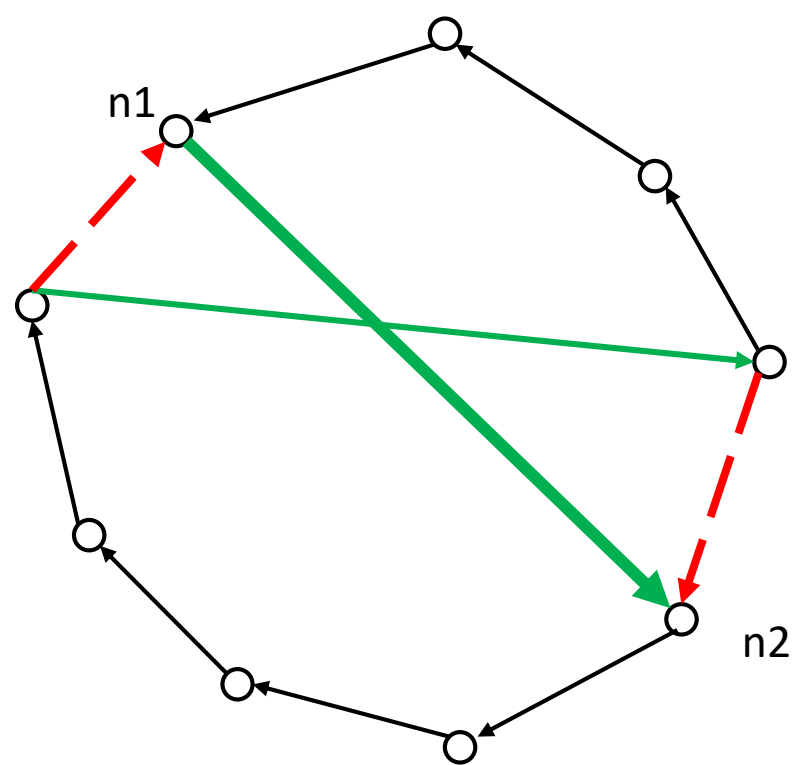
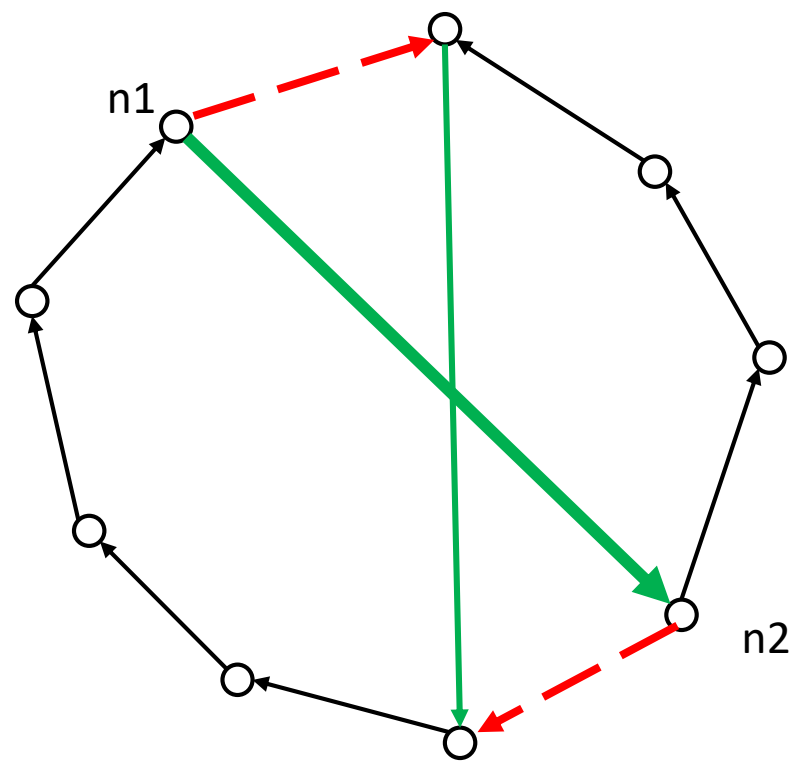
TSP neighborhood search with candidate edges – two-edge exchange – steepest version

For each vertex n_1 from 0 to $N-1$

For each vertex n_2 from the list of the closest vertices to n_1

Evaluate all (two) moves involving the addition of edge n_1-n_2 and the removal of one of the edges adjacent to n_1

Two possible moves



Learning candidate moves

- Learning from a set of solutions:
 - Generate a number of solutions using a heuristic or local search without candidate moves
 - Any edge that occurs at least once (or a certain number/percent of times) in one of these solutions becomes a candidate edge
- The use of higher level methods in which LS is immersed
- Long-term memory e.g.
 - List of good edges in TSP
 - List of edges in accepted/improving moves
- Use of information from recombination operators
 - For example candidate edges include edges that occur in either parent (even if they are not in the offspring)

Advanced techniques of improving LS efficiency

- For example searching a neighborhood of exponential size in polynomial time

Generate an initial solution \mathbf{x}

repeat

find the best move $m \in M(\mathbf{x})$

if $f(m(\mathbf{x})) > f(\mathbf{x})$ **then**

$\mathbf{x} := m(\mathbf{x})$

until no better solution was found after checking the whole $M(\mathbf{x})$

← Optimization problem

- In some cases the use of dynamic programming allows to select the best move from an exponential neighborhood in pseudo polynomial time

Advanced (future) techniques

- The use of extensions of the quantum Grover algorithm for black-box optimization while searching for the best (or improving) move
- $\Theta(\sqrt{|M(\mathbf{x})|})$ eg. $O(n)$ instead of $O(n^2)$ for two edges exchange in TSP

Disadvantages of local search

- Ends up search in a local optimum that doesn't have to be an (global) optimum
 - Unless the neighborhood includes all solutions which leads to a exhaustive search
- The quality of local optima depends on the starting solutions

How to avoid the disadvantages of local search?

- Change/extension of the neighborhood definition to search more solutions
- Accepting to some extent solutions that deteriorate the current solution
- Running LS from different starting points
- Generating good starting solutions
 - Often (usually?) a better starting solution results in both a better local optimum as well as a shorter running time but it can also bias the search

Multiple start local search (MSLS)

Repeat

Generate a randomized starting solution \mathbf{x}

Local search (\mathbf{x})

Until stopping conditions are met

Return the best solution found

- **Remark!** A single MSLS run includes multiple runs of LS

Multiple start local search (MSLS)

- The simplest (vanilla) LS extension
- It should be the baseline for all more complex methods that should prove to be better than MSLS

Adaptive multiple start local search

- After each LS run the frequency table of individual solution elements (e.g. edges) in local optima is updated
- New starting solutions are created using this data – a higher probability of selecting frequent elements (e.g. edges)
 - For example a randomized structural heuristic which when assessing the inserted elements takes into account the frequency of their occurrence

Adaptive multistart local search - example

Effective neighborhood search with optimal splitting and adaptive memory for the team orienteering problem with time windows Youcef Amaroucheab Rym Nesrine Guibadjc Elhadja Chaalalb Aziz Moukrim Computers & Operations Research Volume 123 November 2020 105039

- Team orienteering problem with time windows – type of vehicle routing problem
- A large set of various routes is created from which new starting solutions are built
- Connected with ILS

Variable neighborhood Local Search

- Motivation
 - A local optimum for a certain neighborhood doesn't have to be a local optimum for another type of neighborhood – so it can still be improved by using a different type of neighborhood.
- The global optimum is the local optimum for all possible neighborhoods

Variable neighborhood Local Search

Input – list of (definitions of) neighborhoods (N_1, N_2, \dots)

Generate an initial solution \mathbf{x}

For each neighborhood $N=(N_1, N_2, \dots)$

$\mathbf{x} := \text{Local search } (\mathbf{x}, N)$

Variable neighborhood Local Search

- Usually we start with less complex neighborhoods and move on to more complex ones e.g. for TSP first two edge exchange then three edge exchange etc.
- Typical impact of neighborhood size (although there are other factors):
 - Less complex: shorter running time worse local optimum
 - More complex: longer calculation time better local optimum
- Thanks to the above approach we quickly get a relatively good solution (without wasting time for testing many neighboring solutions) which we can further improve in a relatively small number of steps of LS with a more complex neighborhood.
- We may also return to a previous neighborhood

Iterated local search

Generate an initial solution \mathbf{x}

$\mathbf{x} := \text{Local search}(\mathbf{x})$

Repeat

$\mathbf{y} := \text{Perturb}(\mathbf{x})$

$\mathbf{y} := \text{Local search}(\mathbf{y})$

If $f(\mathbf{y}) > f(\mathbf{x})$ **then**

$\mathbf{x} := \mathbf{y}$

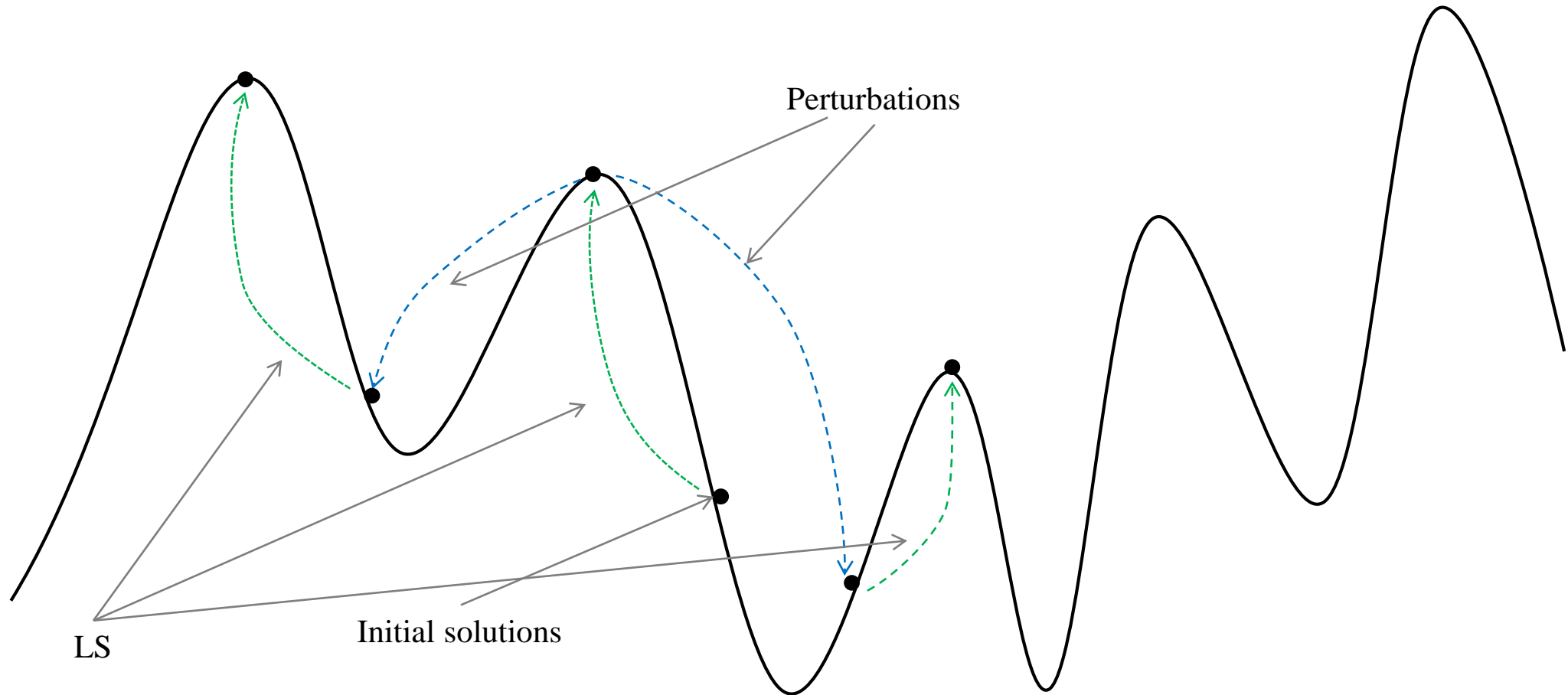
Until stopping conditions are met

- In a sense a local search in a set of local optima (defined with the base neighborhood) using a larger randomly sampled neighborhood

The concept of perturbation

- Randomized modification going beyond the neighborhood used in LS
 - unattainable in one move e.g.:
 - Composition of several moves
 - Larger move e.g. exchanging more edges for TSP
- Perturbation size
 - Too small causes ILS to return to the same local optimum
 - Too large worsens the quality of starting solutions of LS in the extreme case it means creating random solutions

Iterated local search



Large neighborhood search (LNS)

- A move is composed of two methods:
 - Destroy – removing some elements from the solution
 - For example removes some vertices from VRP problem routes
 - Often combining randomness with heuristic choice – we try to remove vertices edges fragments of routes that may be wrongly assigned e.g. different in different solutions
 - Repair –objective function-guided repair of the solution
 - E.g. greedy heuristic inserting previously removed vertices/edges
- In general such a move may cause large changes in the solution

Large neighborhood search

Generate an initial solution \mathbf{x}

$\mathbf{x} := \text{Local search } (\mathbf{x})$ (*optional*)

Repeat

$\mathbf{y} := \text{Destroy } (\mathbf{x})$

$\mathbf{y} := \text{Repair } (\mathbf{y})$

$\mathbf{y} := \text{Local search } (\mathbf{y})$ (*optional*)

If $f(\mathbf{y}) > f(\mathbf{x})$ **then**

$\mathbf{x} := \mathbf{y}$

Until stopping conditions are met

Adaptive large neighborhood search

- We can use many different types of destroy and repair methods
- Their probability is automatically modified based on the effects of individual operators (e.g. the probability of obtaining a new better solution)
- A kind of hyperheuristic

Hyperheurists

- In any iterative heuristic we may use different operators (e.g. neighborhood, perturbation, destroy/repair recombination...)
- Their probability is automatically modified based on the effects of using individual operators (e.g. the probability of obtaining a new better solution)
- Could be done with simple statistical rules or more advanced AI techniques e.g. reinforcement learning.

Stopping conditions

- Number of iterations / running time
- Dynamic condition – no improvement in a given number of iterations / running time

Simulated annealing

- Physical annealing
 - Rapid rise of a solid temperature near (but below) the melting point
 - At a high temperatures atoms/molecules easily change their position in the crystal structure
 - Slow lowering of the temperature
 - When lowering the temperature it becomes increasingly difficult for atoms/molecules to change their position in the crystal structure
 - Leads to the arrangement of atoms/molecules in a regular crystalline structure – a state with minimal energy

Physical annealing – analogies with optimization

- Structure of atoms/molecules arrangement – solution
- Energy – objective function (minimized)
- Ideal crystalline structure – global optimum
- Temperature – control parameter
- Rapid cooling – local optimization
- Slow cooling – simulated annealing

Simulated annealing

Generate an initial solution \mathbf{x}

Temperature $T := T_0$

repeat

repeat L times

 Generate random $\mathbf{y} \in N(\mathbf{x})$

if $f(\mathbf{y}) > f(\mathbf{x})$ **then**

$\mathbf{x} := \mathbf{y}$

else

if $\exp((f(\mathbf{y}) - f(\mathbf{x})) / T) > \text{random}[0,1)$

$\mathbf{x} := \mathbf{y}$

 Lower T

until $T \leq T_k$

Return the best generated solution

Acceptance condition

- Better solutions are always accepted
- Worse solutions are accepted with some probability
 - The smaller the deterioration of the objective function value the greater the probability
 - The higher the temperature T the greater the probability

Very high temperature – random walk

Very low temperature – local search

SA – in between

Convergence of simulated annealing

- With a sufficiently high initial temperature and a sufficiently slow decreasing of temperature the probability of obtaining the global optimum tends to 1
- The result is rather theoretical in practice the required number of moves is similar to the exhaustive search

A typical way of decreasing the temperature

$$T_{k+1} = \alpha T_k \quad k = 1, 2 \dots$$

$$T_{k+1} = \alpha^k T_0$$

α is a constant smaller but close to 1 typically [0.8 – 0.99]

Initial temperature

- Traditional approach (motivated by the theoretical convergence)
 - The initial temperature should be high enough to ensure acceptance of majority of moves
 - If too low raising the temperature
 - Depends on a typical change in the objective function value Δf
 - E.g. for $\Delta f = 1000$ and probability of acceptance 0.98, $T_0 \approx 250$
- In practice better results are often obtained starting from much lower temperatures.

Final temperature

- Sufficiently low probability of acceptance

Great Deluge (Deterministic SA?)

Generate an initial solution \mathbf{x}

Water level $P := P_0$

repeat

repeat L times

 Generate random $\mathbf{y} \in N(\mathbf{x})$

if $f(\mathbf{y}) > P$ **then**

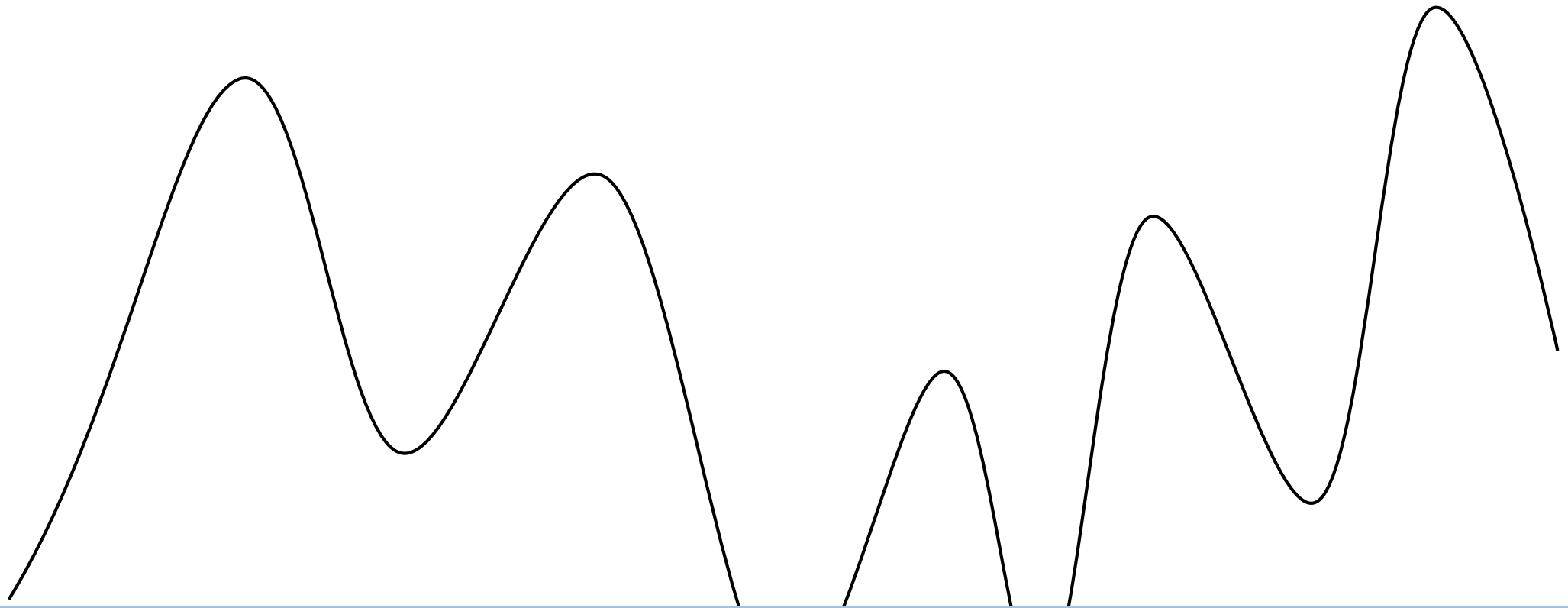
$\mathbf{x} := \mathbf{y}$

 increase P

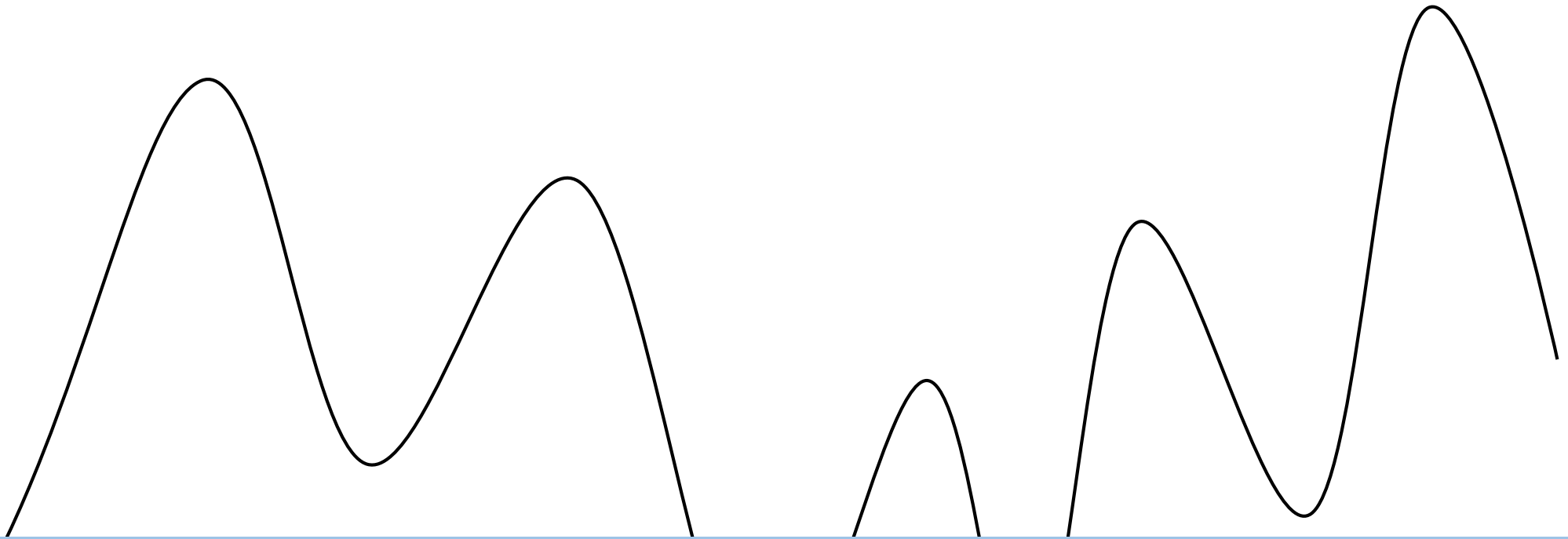
until $P \leq P_K$

Return the best generated solution

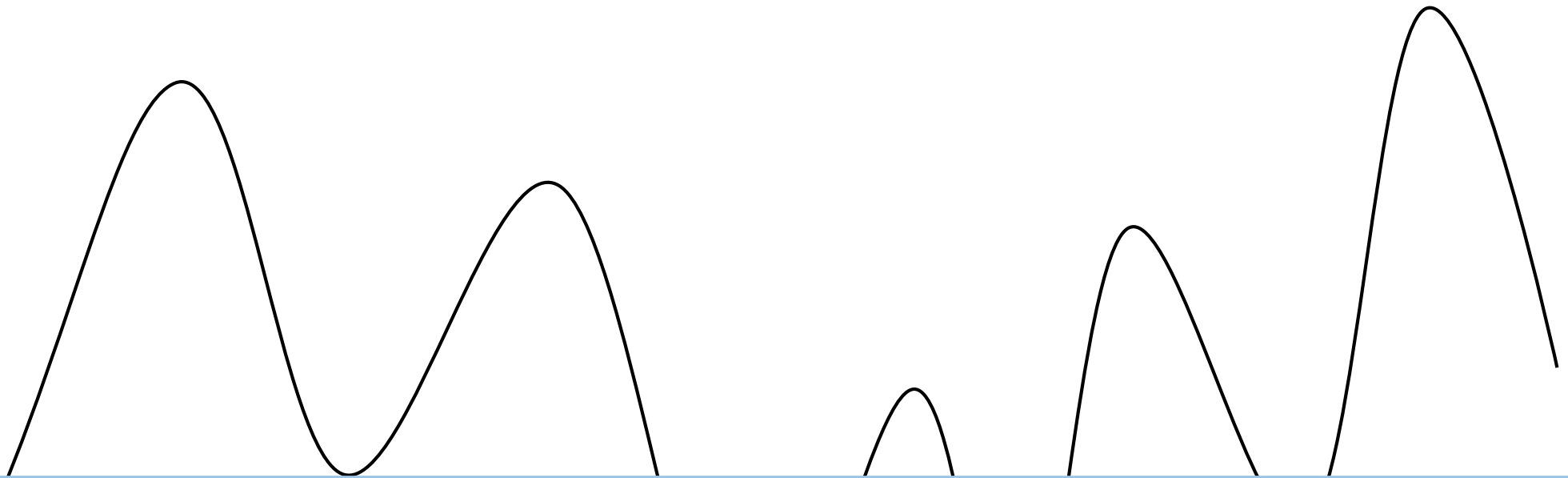
Great Deluge



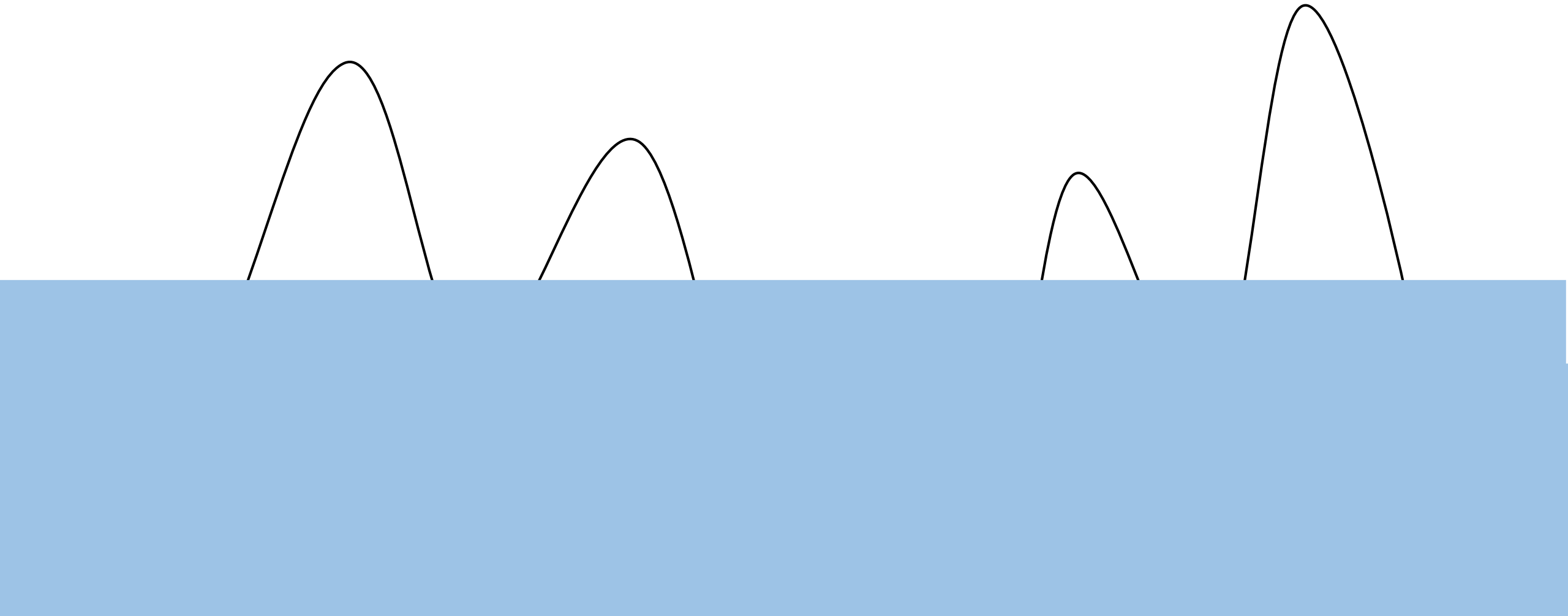
Great Deluge



Great Deluge



Great Deluge



Tabu search

- The starting point is LS in steepest version
- The question is whether it is possible to abandon the condition that the new solution must be better than the current one?
- This however leads to the emergence of cycles (returns to the same solutions)

Generate an initial solution \mathbf{x}

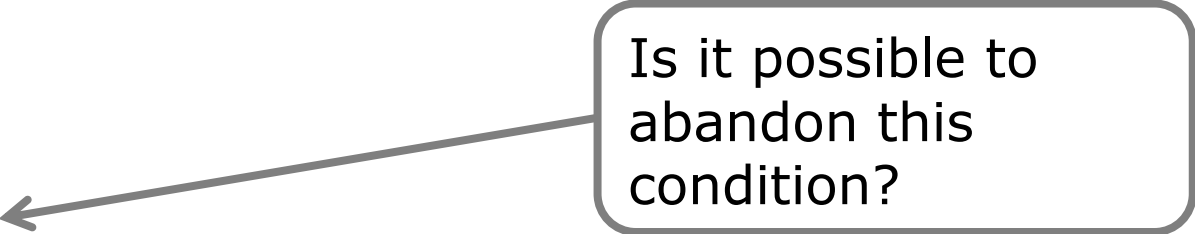
repeat

Find the best move $m \in M(\mathbf{x})$

if $f(m(\mathbf{x})) > f(\mathbf{x})$ **then**

$\mathbf{x} := m(\mathbf{x})$

until no better solution has been found



Is it possible to
abandon this
condition?

Cycling problem

- A simple elimination of the above condition would lead to cycles – returns to the same solutions
- The easiest way to avoid cycles:
 - Remember all solutions visited so far
 - These solutions become prohibited – Tabu
 - For improved efficiency hash values and binary search
 - Still may lead to „cycling" over very similar solutions

The idea of a tabu list

- List of prohibited solutions/moves/move elements

Algorithm of Tabu search

Generate an initial solution \mathbf{x}

Tabu list $T := \emptyset$

repeat

Find the best move $m \in M(\mathbf{x}) \mid m \notin T$

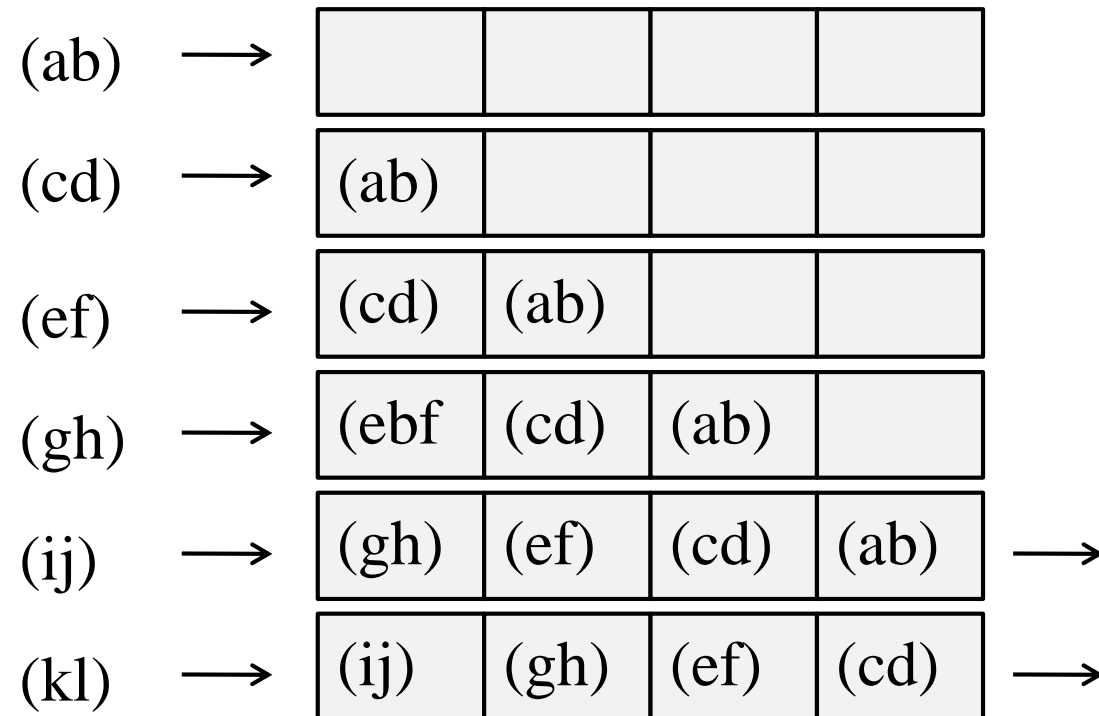
$\mathbf{x} := m(\mathbf{x})$

Update Tabu list (T, m)

until stopping condition is met

Return the best generated solution

Tabu list as a FIFO queue



Design options

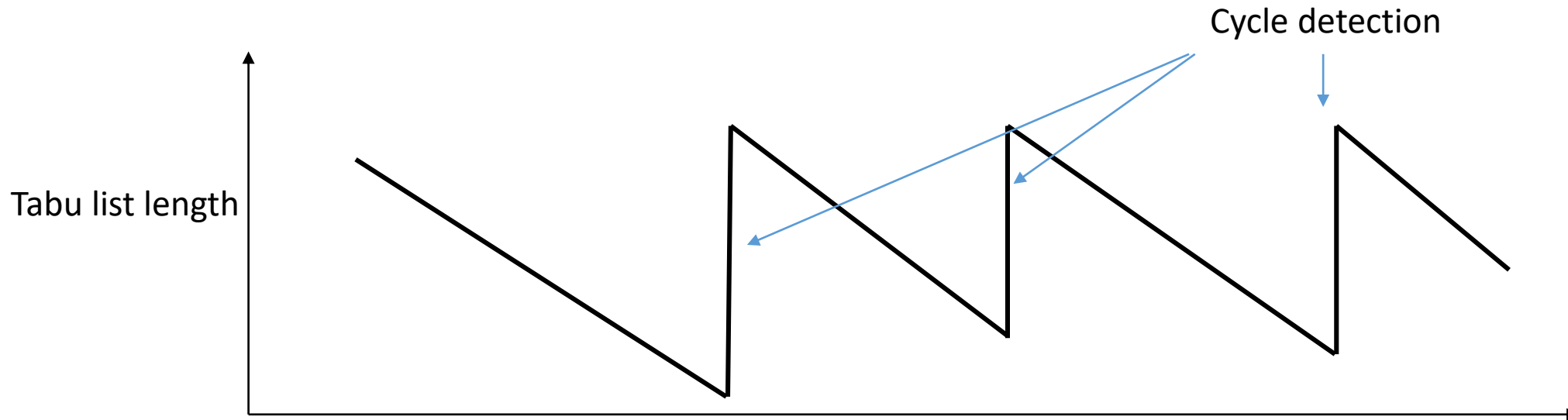
- What to store in the Tabu list?
 - Solutions (see earlier notes)
 - Moves
 - Components of moves (e.g. single edges vertices)
- How to block moves?
 - Exact match
 - Only moves in the Tabu list or inverse moves are prohibited
 - Relatively small range of blocking
 - Requires a longer Tabu List
 - Partial match
 - For example stored move (ab) blocks all moves in which a or b occurs
 - Natural when storing the component of moves
 - Relatively large range of blocking
 - Requires a shorter Tabu list
- Tabu list length

Tabu list length

- Short list
 - More "aggressive" algorithm
 - Higher risk of falling into a cycle
- Long list
 - Lower risk of falling into a cycle
 - Risk of blocking too many moves
 - It may even prevent TS from reaching any local optimum

Reactive Tabu Search – automatic modification the Tabu list length

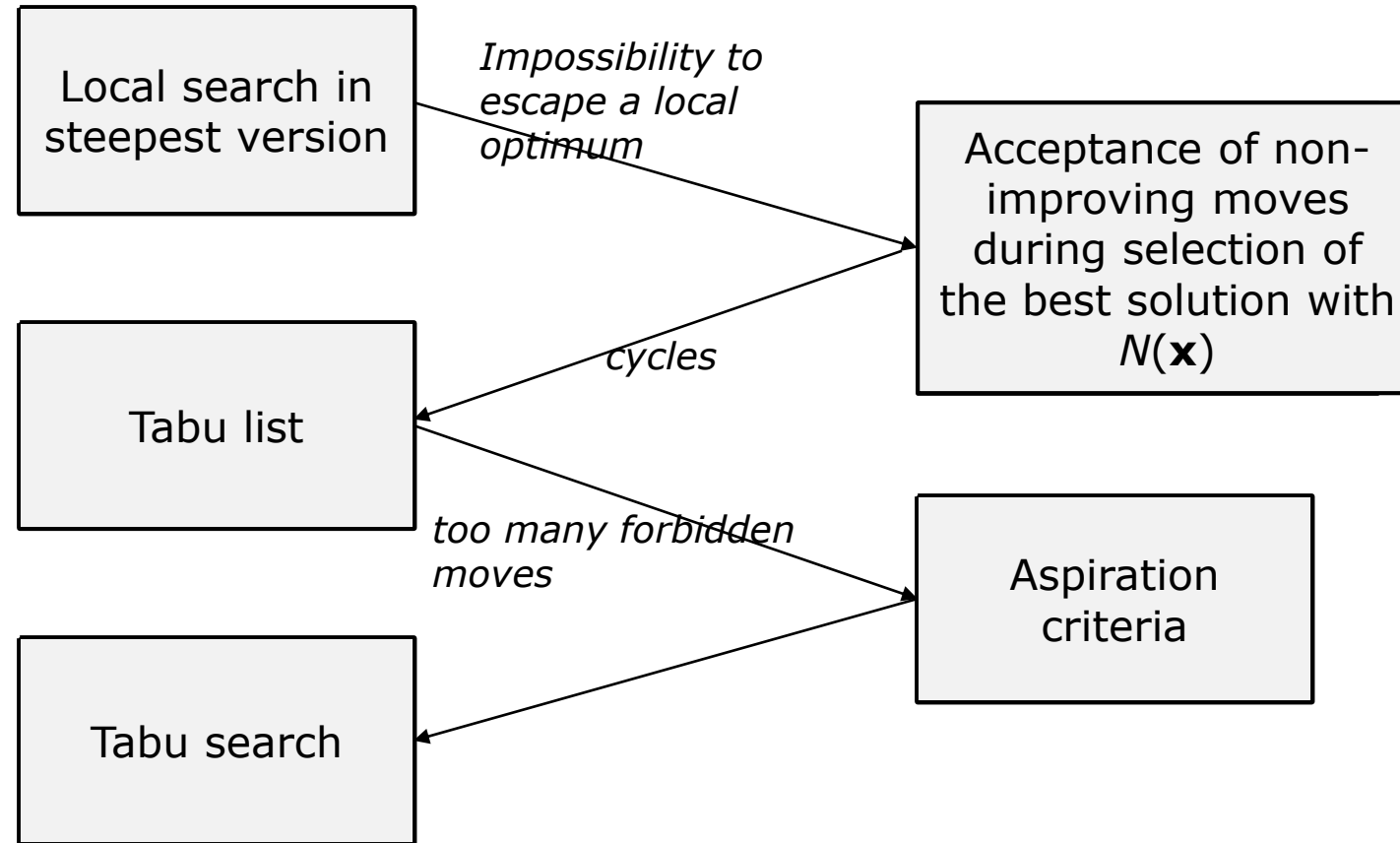
- Constant slow reduction of the length of a list
 - Every L iteration reduction of the length of the Tab list
- Sharp increase of the length of the list when falling into a cycle
 - Requires storing all visited solutions to discover a cycle



Aspiration criteria

- Tabu list may block too many moves/solutions
- Tabu moves/solutions that are very good – i.e. meet the aspiration criteria – can be accepted
- The most obvious aspiration criterion – accept solutions better than the best found so far even if they are Tabu
 - Such solutions have certainly not been visited before
- One can also accept relatively good solutions (but not better than the best) significantly different from the previous ones.

The natural way of creating the Tabu search algorithm



Long term memory

- A concept independent of Tabu search although originally proposed in this context
- Examples:
 - Table of frequency of occurrences of individual edges in the visited solutions
 - Table of frequency of improving moves
- Applications:
 - Restarts – creating new solutions:
 - Diversification – starting solutions far from those visited so far
 - Intensification – starting solutions similar to the best ones visited so far
 - Defining candidate moves

Combining/hybridizing different methods based on LS

- E.g. simulated annealing, Tabu search, local search with variable neighborhood as a part of iterated local search
- Iterated local search, simulated annealing, Tabu search as part of a Large neighborhood search
- ILS with acceptance criterion of simulated annealing
- etc...