

# Optimization and Simulation

## Optimization

Michel Bierlaire

Transport and Mobility Laboratory  
School of Architecture, Civil and Environmental Engineering  
Ecole Polytechnique Fédérale de Lausanne



**EPFL**

# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms
  - Brute force
  - Greedy heuristics
  - Exploration
  - Intensification
  - Diversification
- 4 Summary

# Optimization

## Procedure

- Mathematical modeling.
- Selection of an algorithm.
- Solving the problem.

# Optimization

## Mathematical modeling

- Decision variables  $x$ .
- Objective function  $f$ .
- Constraints  $X$ .

## Optimization problem

$$\min_{x \in \mathbb{R}^n} f(x)$$

subject to

$$x \in X \subseteq \mathbb{R}^n.$$

# Optimization

## Selection of an algorithm

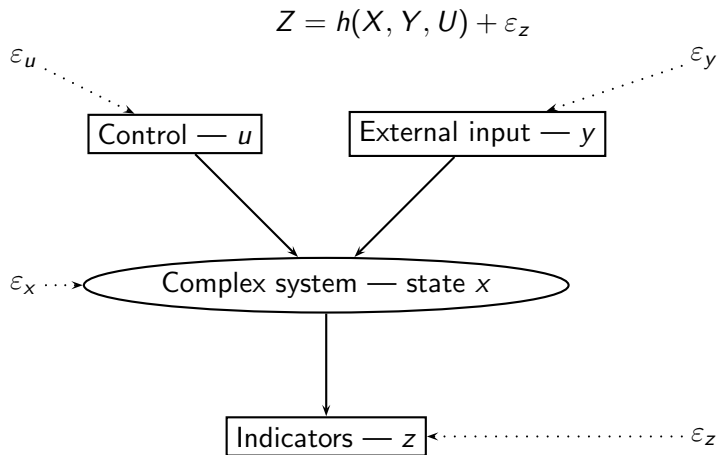
- Mathematical properties of the model.
- Linear optimization.
- Convex optimization.
- Mixed integer linear optimization.
- Differentiable optimization.

# Optimization

## Solving the problem

- Implement, or obtain the code of the algorithm.
- Import the data.
- Solve.

# General framework



# General framework

## Assumptions

- Control  $U$  is deterministic.

$$Z(u) = h(X, Y, u) + \varepsilon_Z$$

- Various features of  $Z$  are considered: mean, variance, quantile, etc.

$$(z_1(u), \dots, z_m(u))$$

- They are combined in a single indicator:

$$f(u) = g(z_1(u), \dots, z_m(u))$$

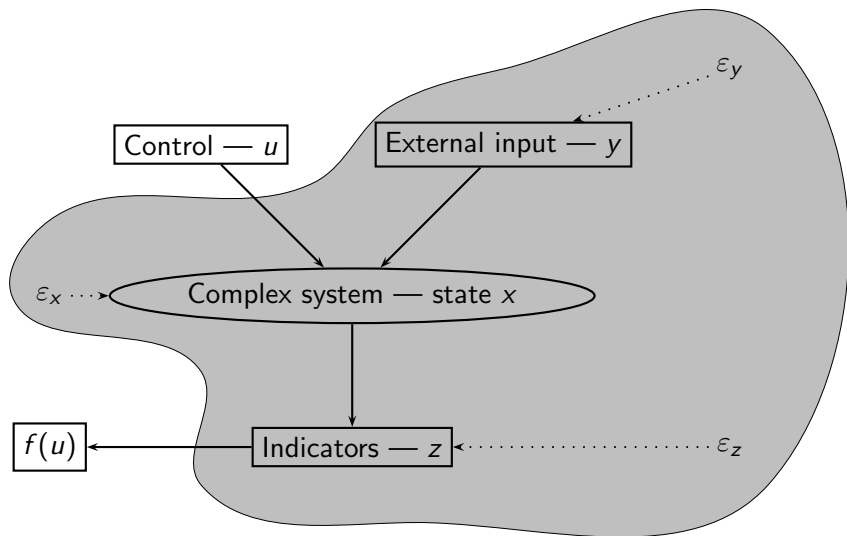


# General framework: example

## Cloe at Satellite

- $X$ : number of customers in the bar
- $Y$ : arrivals of customers
- $u$ : service time of Cloe
- $Z(u)$ : waiting time of the customers
- $z_1(u)$ : mean waiting time
- $z_2(u)$ : maximum waiting time
- $f(u) = g(z_1(u), z_2(u)) = z_1 + z_2$

## General framework: the black box



# Optimization problem

$$\min_{u \in \mathbb{R}^n} f(u)$$

subject to

$$u \in \mathcal{U} \subseteq \mathbb{R}^n$$

- $u$ : decision variables
- $f(u)$ : objective function
- $u \in \mathcal{U}$ : constraints
- $\mathcal{U}$ : feasible set

# Optimization problem

## Combinatorial optimization

- $f$  and  $\mathcal{U}$  have no specific property.
- $f$  is a black box.
- $\mathcal{U}$  is a finite set of valid configurations.
- No optimality condition is available.

# Optimization methods

## Exact methods (branch and bound)

- Finds the optimal solution.
- Suffers from the curse of dimensionality.
- Requires the availability of valid and tight bounds.

## Approximation algorithms

- Finds a sub-optimal solution.
- Guarantees a bound on the quality of the solution.
- Mainly used for theoretical purposes.

## Heuristics

- Smart exploration of the solution space.
- No guarantee about optimality.
- Few assumptions about the problem.

# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms
  - Brute force
  - Greedy heuristics
  - Exploration
  - Intensification
  - Diversification
- 4 Summary

# The knapsack problem

- Patricia prepares a hike in the mountain.
- She has a knapsack with capacity  $W$ kg.
- She considers carrying a list of  $n$  items.
- Each item has a utility  $u_i$  and a weight  $w_i$ .
- What items should she take to maximize the total utility, while fitting in the knapsack?



# Mathematical model

## Decision variables

$$x_i = \begin{cases} 1 & \text{if item } i \text{ goes into the knapsack,} \\ 0 & \text{otherwise} \end{cases}$$

## Objective function

$$\max f(x) = \sum_{i=1}^n u_i x_i$$

## Constraints

$$\sum_{i=1}^n w_i x_i \leq W$$

$$x_i \in \{0, 1\} \quad i = 1, \dots, n$$



# Instance

$$n = 12$$

Maximum weight: 300.

Item	Utility	Weight
1	80	84
2	31	27
3	48	47
4	17	22
5	27	21
6	84	96
7	34	42
8	39	46
9	46	54
10	58	53
11	23	32
12	67	78

# Real example



## Portfolio optimization

- Items: potential assets.
- Utility: return.
- Weight: risk.
- Capacity: maximum risk.

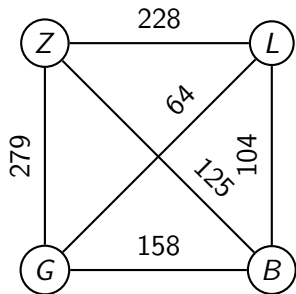
# Traveling salesman problem

## The problem

- Consider  $n$  cities.
- For any pair  $(i, j)$  of cities, the distance  $d_{ij}$  between them is known.
- Find the shortest possible itinerary that starts from the home town of the salesman, visit all other cities, and come back to the origin.

## TSP: example

Lausanne, Geneva, Zurich, Bern

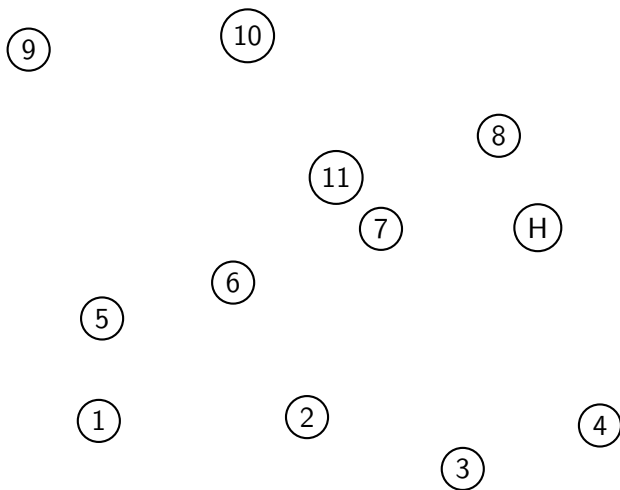


Home town: Lausanne

3 possibilities (+ their symmetric version):

- $L \rightarrow B \rightarrow Z \rightarrow G \rightarrow L$ : 572 km
- $L \rightarrow B \rightarrow G \rightarrow Z \rightarrow L$ : 769 km
- $L \rightarrow Z \rightarrow B \rightarrow G \rightarrow L$ : 575 km

## TSP: 12 cities (euclidean dist.)



# Integer linear optimization problem

## Linear optimization

$$\min_{x \in \mathbb{R}^n} c^T x$$

subject to

$$Ax = b$$

$$x \geq 0.$$

where  $A \in \mathbb{R}^{m \times n}$ ,  $b \in \mathbb{R}^m$  and  $c \in \mathbb{R}^n$ .

## Integer Linear optimization

$$\min_{x \in \mathbb{R}^n} c^T x$$

subject to

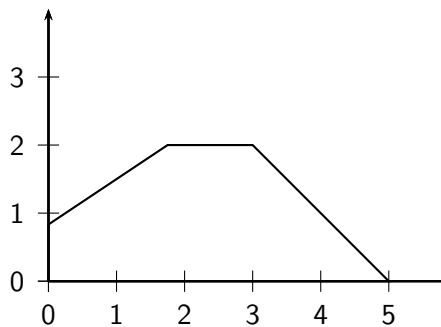
$$Ax = b$$

$$x \in \mathbb{N}.$$

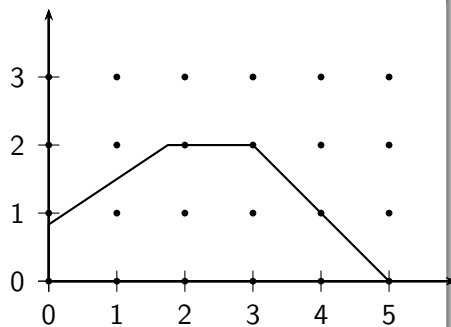
where  $A \in \mathbb{R}^{m \times n}$ ,  $b \in \mathbb{R}^m$  and  $c \in \mathbb{R}^n$ .

## Feasible set

Polyhedron



Intersection polyhedron/integer lattice



# Example

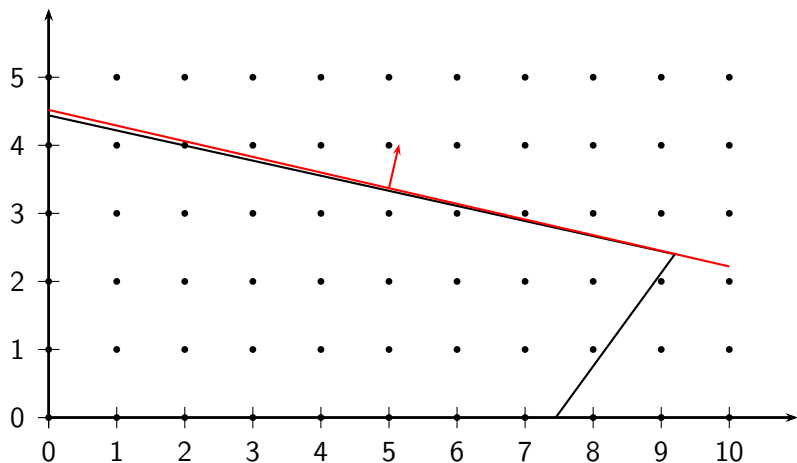
$$\min_{x \in \mathbb{R}^2} -3x_1 - 13x_2$$

subject to

$$\begin{aligned} 2x_1 + 9x_2 &\leq 40 \\ 11x_1 - 8x_2 &\leq 82 \\ x_1, x_2 &\geq 0 \\ x_1, x_2 &\in \mathbb{N} \end{aligned}$$



## Example



# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms**
  - Brute force
  - Greedy heuristics
  - Exploration
  - Intensification
  - Diversification
- 4 Summary

# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms
  - Brute force
  - Greedy heuristics
  - Exploration
  - Intensification
  - Diversification
- 4 Summary

# Brute force algorithm

$$\min_{u \in \mathbb{R}^n} f(u)$$

subject to

$$u \in \mathcal{U} \subseteq \mathbb{R}^n$$

## Brute force algorithm

- $f^* = +\infty$
- For each  $x \in \mathcal{U}$ , if  $f(x) < f^*$  then  $x^* = x$ ,  $f^* = f(x^*)$ .

# Knapsack problem

## Enumeration

- Each object can be in or out, for a total of  $2^n$  combinations.
- For each of them, we must:
  - Check that the weight is feasible.
  - If so, calculate the utility and check if it is better than  $f^*$ .

# Python implementation

```
import numpy as np
import itertools
utility = np.array([80, 31, 48, 17, 27, 84, 34, 39, 46, 58, 23, 67])
weight = np.array([84, 27, 47, 22, 21, 96, 42, 46, 54, 53, 32, 78])
capacity = 300
n = len(utility)
fstar = -np.inf
xstar = None
for c in itertools.product([0, 1], repeat = n):
    w = np.inner(c, weight)
    if w <= capacity:
        u = np.inner(c, utility)
        if u > fstar:
            xstar = c
            fstar = u
```

Solution: (1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0). Weight: 300. Utility: 300.

# Knapsack problem

## Computational time

- About  $2n$  floating point operations per combination.
- Assume a 1 Teraflops processor:  $10^{12}$  floating point operations per second.

# Knapsack problem

## Computational time

- If  $n = 34$ , about 1 second to solve.
- If  $n = 40$ , about 1 minute.
- If  $n = 45$ , about 1 hour.
- If  $n = 50$ , about 1 day.
- If  $n = 58$ , about 1 year.
- If  $n = 69$ , about 2583 years, more than the Christian Era.
- If  $n = 78$ , about 1,500,000 years, time elapsed since Homo Erectus appeared on earth.
- If  $n = 91$ , about  $10^{10}$  years, roughly the age of the universe.



# Traveling salesman problem

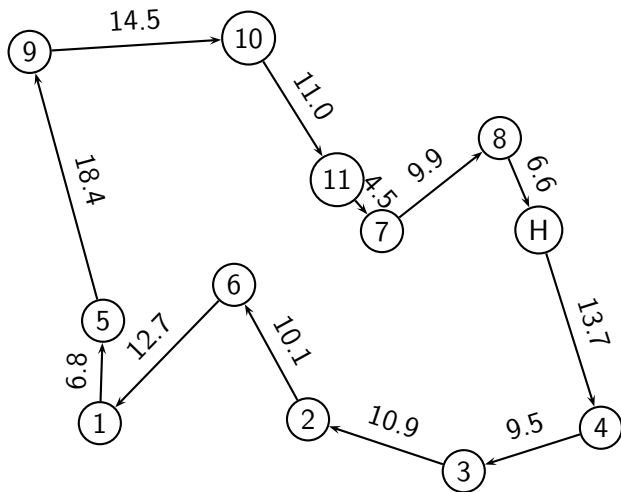
## Python code

```
fstar = np.inf
xstar = None
for t in itertools.permutations(names[1:]):
    tour = ['0']+list(t)
    tl = tsp.tourLength(tour)
    if tl < fstar:
        xstar = tour
        fstar = tl
```

## TSP with 12 cities

- $11! = 39916800$  permutations.
- Running time: about 5 minutes.
- Solution: H-4-3-2-6-1-5-9-10-11-7-8
- Tour length: 128.762

# Optimal solution



Length : 128.762

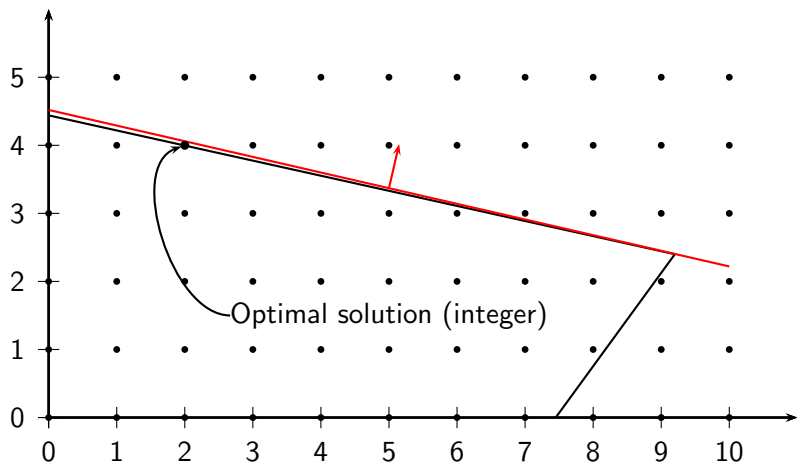
# Integer optimization

$$\min_{x \in \mathbb{R}^2} -3x_1 - 13x_2$$

subject to

$$\begin{aligned} 2x_1 + 9x_2 &\leq 40 \\ 11x_1 - 8x_2 &\leq 82 \\ x_1, x_2 &\geq 0 \\ x_1, x_2 &\in \mathbb{N} \end{aligned}$$

# Feasible set: 36 solutions



# Brute force algorithm

## Comments

- Very simple to implement.
- Works only for small instances.
- Curse of dimensionality.
- Running time increases exponentially with the size of the problem.
- Not a reasonable option.

# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms
  - Brute force
  - Greedy heuristics
  - Exploration
  - Intensification
  - Diversification
- 4 Summary

# Greedy heuristics

## Principles

- Step by step construction of a feasible solution.
- At each step, a local optimization is performed.
- Decisions taken at previous steps are definitive.

## Properties

- Easy to implement.
- Short computational time.
- May generate poor solutions.
- Used to generate initial solutions.

# The knapsack problem

## Greedy heuristic

- Sort the items by decreasing order of  $u_i/w_i$ .
- For each item in this order, put it in the sack if it fits.



# The knapsack problem

Item	Utility	Weight	Ratio
1	80	84	0.952
2	31	27	1.148
3	48	47	1.021
4	17	22	0.773
5	27	21	1.286
6	84	96	0.875
7	34	42	0.810
8	39	46	0.848
9	46	54	0.852
10	58	53	1.094
11	23	32	0.719
12	67	78	0.859

# The knapsack problem

Item	Utility	Weight	Ratio	Order	Remaining capacity
1	80	84	0.952	5	68
2	31	27	1.148	2	252
3	48	47	1.021	4	152
4	17	22	0.773		
5	27	21	1.286	1	279
6	84	96	0.875	6	-28
7	34	42	0.810		
8	39	46	0.848		
9	46	54	0.852		
10	58	53	1.094	3	199
11	23	32	0.719		
12	67	78	0.859		

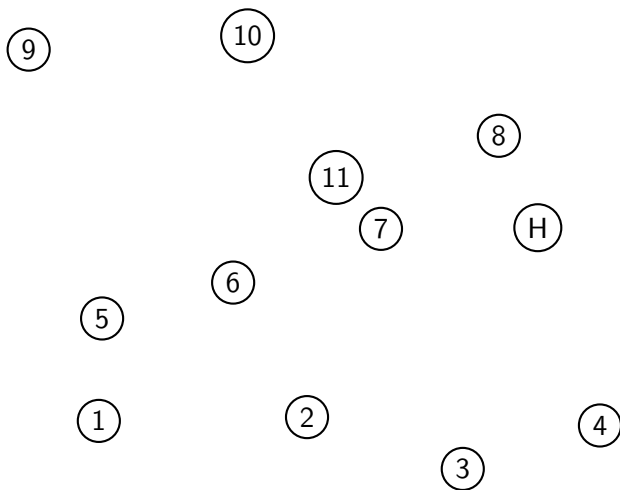
Utility: 244 (Opt: 300). Weight: 232.

# The traveling salesman problem

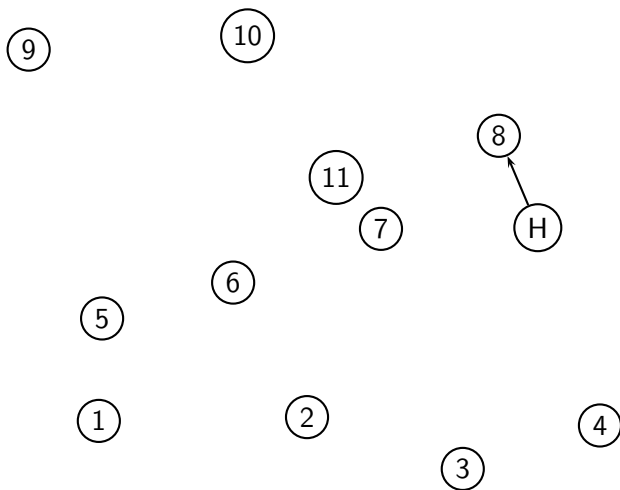
## Greedy heuristic

- Start from home.
- At each step, select the closest city as the next one.

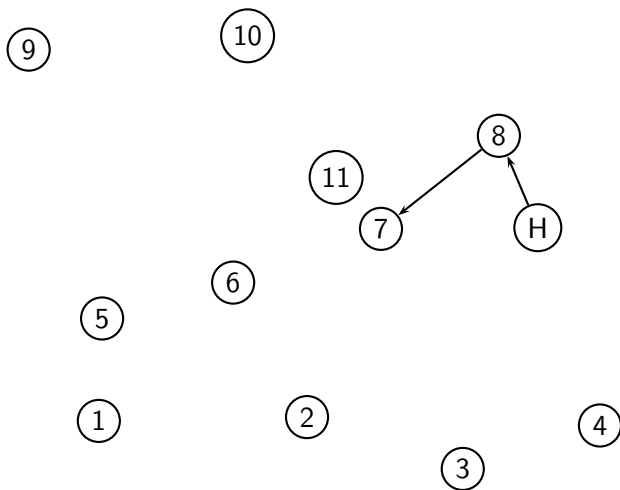
## TSP: 12 cities



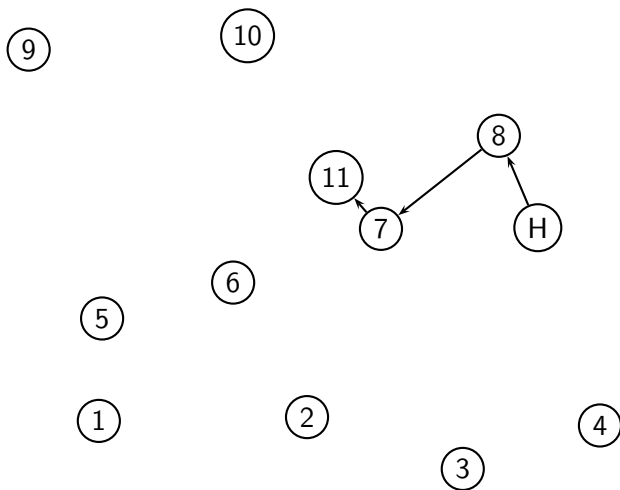
## TSP: 12 cities



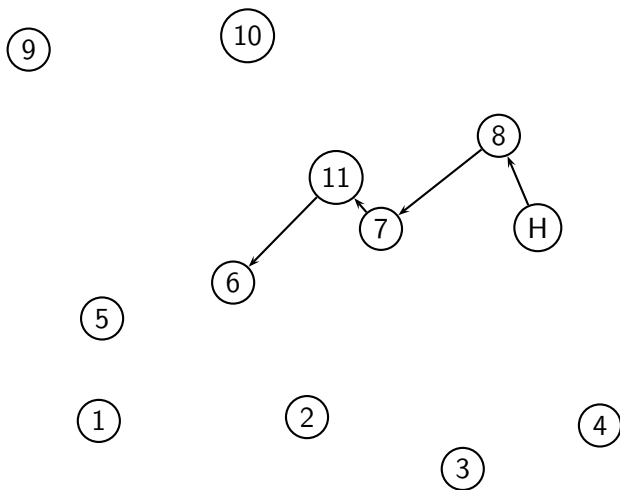
## TSP: 12 cities



## TSP: 12 cities

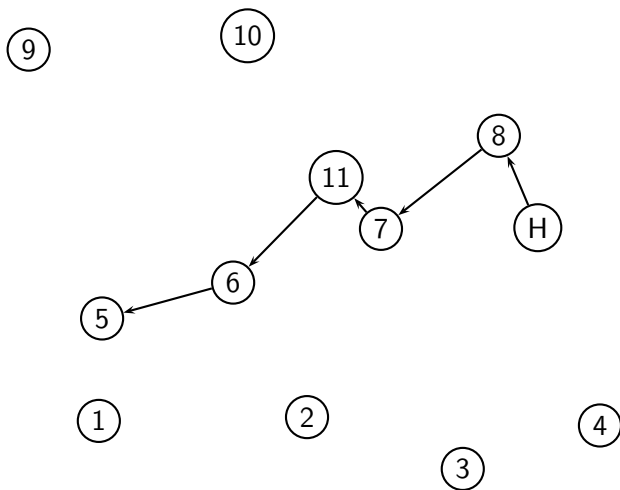


## TSP: 12 cities

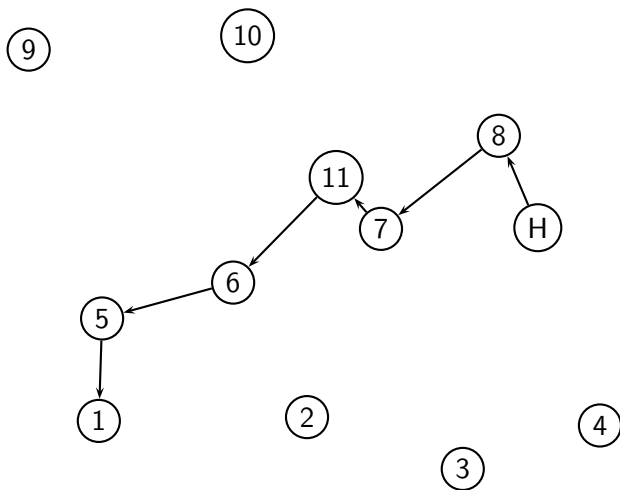




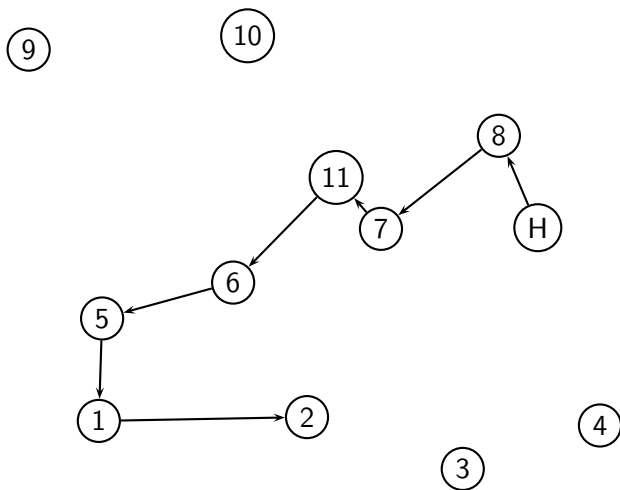
## TSP: 12 cities



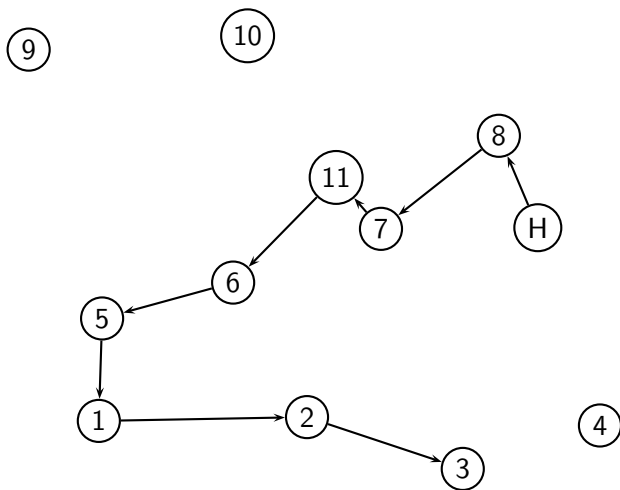
## TSP: 12 cities



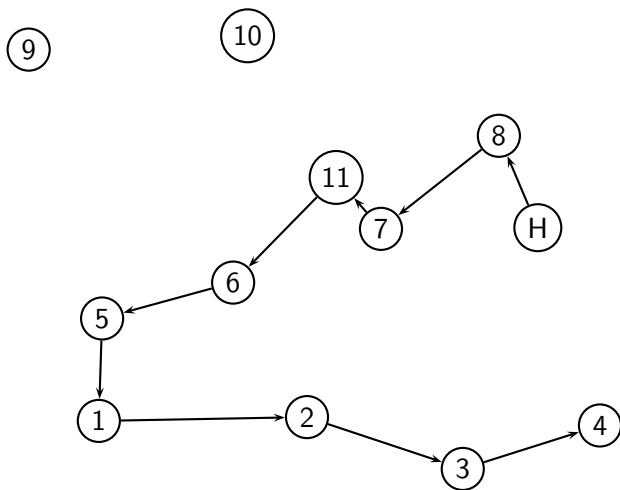
## TSP: 12 cities



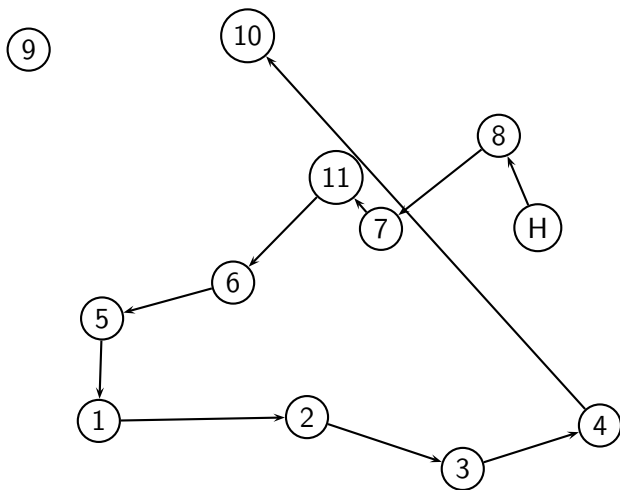
## TSP: 12 cities



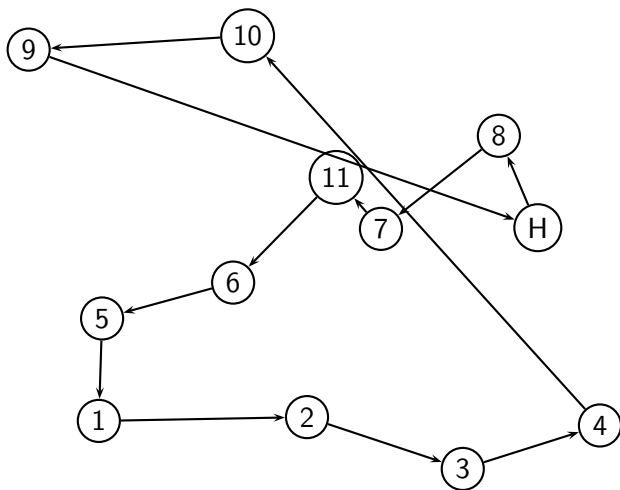
## TSP: 12 cities



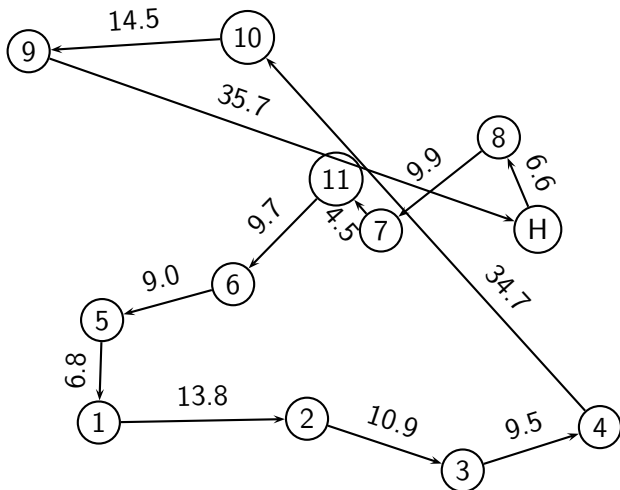
## TSP: 12 cities



## TSP: 12 cities



## TSP: 12 cities





# Integer optimization

## Intuitive approach

- Solve the continuous relaxation.
- Round the solution.

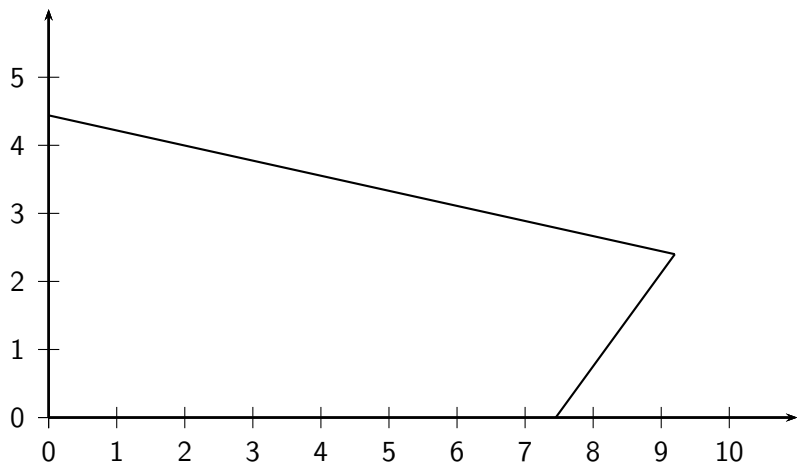
# Example

$$\min_{x \in \mathbb{R}^2} -3x_1 - 13x_2$$

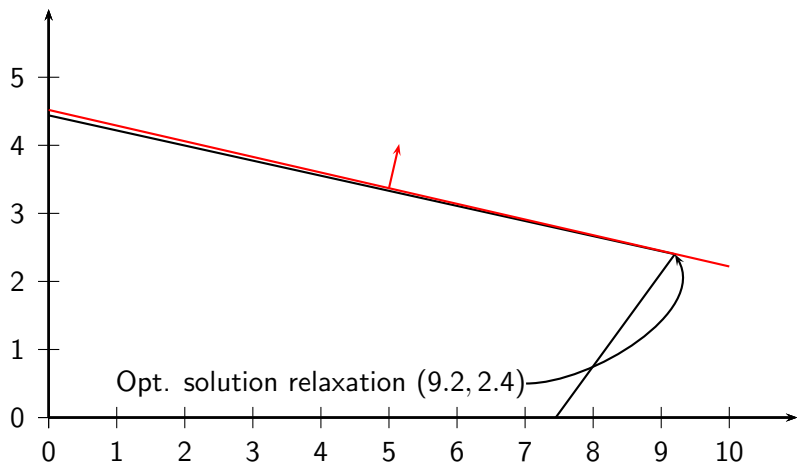
subject to

$$\begin{aligned} 2x_1 + 9x_2 &\leq 40 \\ 11x_1 - 8x_2 &\leq 82 \\ x_1, x_2 &\geq 0 \\ x_1, x_2 &\in \mathbb{N} \end{aligned}$$

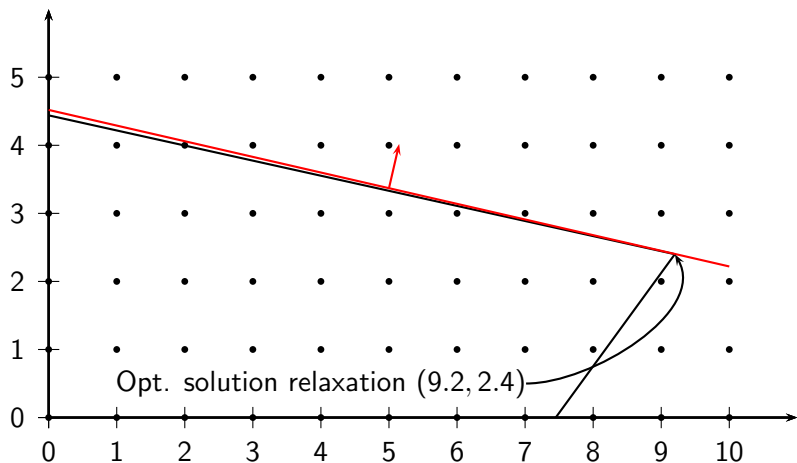
# Relaxation: feasible set



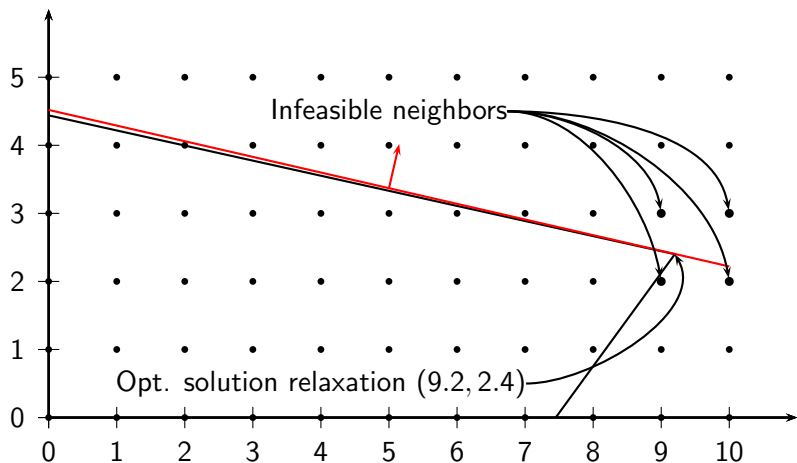
# Optimal solution of the relaxation



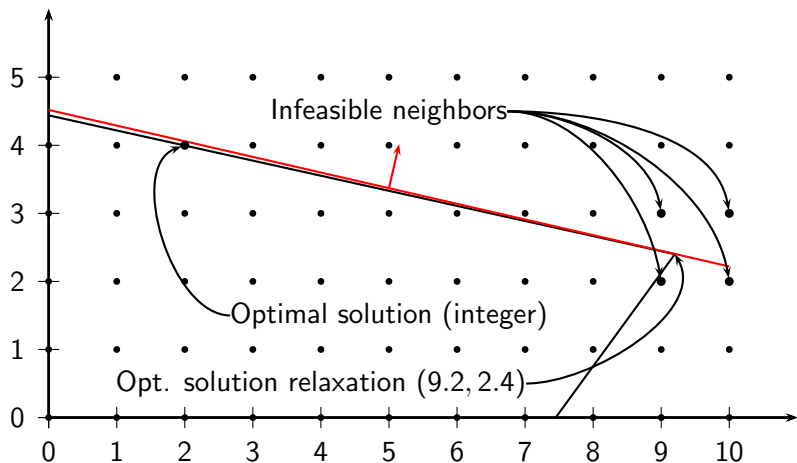
# Integrality constraints



# Infeasible neighbors



# Solution of the integer optimization problem



# Issues

- There are  $2^n$  different ways to round. Which one to choose?
- Rounding may generate an infeasible solution.
- The rounded solution may be far from the optimal solution.



# Greedy heuristics

## Comments

- Fast.
- Easy to implement.
- Useful to find an initial solution.
- Feasibility is usually the main issue (rounding issues with ILP).

# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms
  - Brute force
  - Greedy heuristics
  - **Exploration**
  - Intensification
  - Diversification
- 4 Summary

# Heuristics: general framework

## Exploration

Neighborhood

## Intensification

Local search

## Diversification

Escape from local minima

# Neighborhood

## Concept

- The feasible set is too large.
- We need to explore it in a smart way.
- Idea: at each iteration, restrict the optimization problem to a small feasible subset that can be enumerated.
- The small subset is called a *neighborhood*.
- It is a sorted list of solutions.
- Ideally, all these solutions must be feasible.
- Neighborhoods can be constructed incrementally during the algorithms.

# Neighborhood types

## Fundamental neighborhood structure

- Obtained from simple modifications of the current solution.
- These modifications must be designed based on the properties of the problem.

## Shuffled neighborhood structure

- Obtained from shuffling the solutions from another neighborhood.
- The shuffling can be deterministic or random.

## Feasible neighborhood structure

- Useful when a potential neighborhood structure contains infeasible solutions.
- Feasibility checks can also be done while generating the neighbors.

# Neighborhood types

## Truncated neighborhood structure

- Useful when a potential neighborhood structure is too large.
- The size of the neighborhood is controlled.

## Combined neighborhood structure

- Union, intersection, or any combination of other structures.
- Use building blocks to construct more complex structures.

# Neighborhoods

## Important properties

- Neighborhood structures are used to explore the solution space.
- Algorithms will move from  $x$  to an element of  $V(x)$ .
- They can be seen as “vehicles”.
- Symmetry: it is good practice to use symmetric neighborhoods:

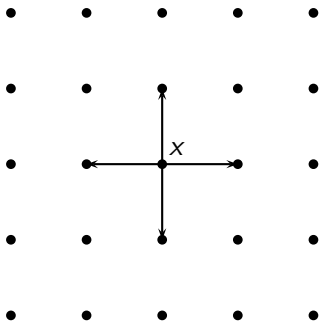
$$y \in V(x) \iff x \in V(y).$$

- Reachability: a neighborhood  $V$  must be rich enough to reach any feasible solution, from another feasible solution. For each  $x_1, x_K \in \mathcal{U}$ , there exists a sequence  $x_2, \dots, x_{K-1} \in \mathcal{U}$  such that

$$x_{k+1} \in V(x_k), \quad k = 1, \dots, K - 1.$$

- Analogy with Markov chains: irreducibility.

# Integer optimization





# Integer optimization

- Consider the current iterate  $x \in \mathbb{Z}^n$ .
- For each  $k = 1, \dots, n$ , define 2 neighbors by increasing and decreasing the value of  $x_k$  by one unit.
- The neighbors  $y^{k+}$  and  $y^{k-}$  are defined as

$$y_i^{k+} = y_i^{k-} = x_i, \forall i \neq k, \quad y_k^{k+} = x_k + 1, \quad y_k^{k-} = x_k - 1.$$

- Example

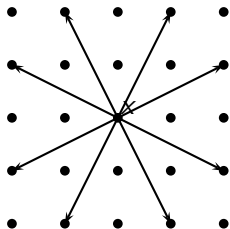
$$x = (3, 5, 2, 8) \quad y^{2+} = (3, 6, 2, 8) \quad y^{2-} = (3, 4, 2, 8)$$

- Size of the neighborhood:  $2n$ .
- Feasibility should also be enforced.
- If  $n$  is large, truncation may be useful.
- The order is arbitrary, but must be specified.
- Shuffling may be useful.

# Integer optimization

## Creativity

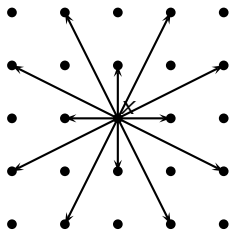
- The concept of neighborhood is fairly general.
- It must be defined based on the structure of the problem.
- Creativity is required here.



# Integer optimization

## Combinations

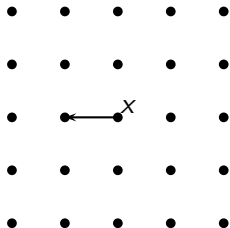
- Combining neighborhoods is easy.
- Trade-off between flexibility and complexity.



# Integer optimization

## Properties

- Verify the properties.
- Symmetry and reachability.



# The knapsack problem

## Fundamental neighborhood

- Current solution: for each item  $i$ ,  $x_i = 0$  or  $x_i = 1$ .
- Neighbor solution: select an item  $j$ , and change the decision:  
 $x_j \leftarrow 1 - x_j$ .
- Warning: check feasibility.
- Generalization: neighborhood of size  $k$ : select  $k$  items, and change the decision for them (checking feasibility).
- Order: based on the utility/weight ratio, for instance.

# The knapsack problem

## Truncated neighborhood

- A neighborhood of size  $k$  modifies  $k$  variables.
- Number of neighbors:

$$\frac{n!}{k!(n-k)!}$$

- $k = 1$ :  $n$  neighbors.
- $k = n$ : 1 neighbor.
- Useful to truncate to  $M$ .
- Size of the neighborhood:

$$\min\left(\frac{n!}{k!(n-k)!}, M\right).$$

# Python code

```
def neighborhood(sack, size = 1, random = True, truncated = None):
    n = len(sack)
    combinations = np.array(list(itertools.combinations(range(n), size)))
    if random:
        np.random.shuffle(combinations)
    if truncated is not None:
        combinations = combinations[:truncated]
    theNeighborhood = []
    for c in combinations:
        s = np.array(sack)
        s[c] = 1 - sack[c]
        theNeighborhood.append(s)
    return theNeighborhood
```

# Traveling salesman problem

## 2-OPT

- Select two cities.
- Swap their position in the tour.
- Visit all intermediate cities in reverse order.

## Example

Current tour:

A-B-C-D-E-F-G-H-A

Exchange C and G to obtain

A-B-G-F-E-D-C-H-A.

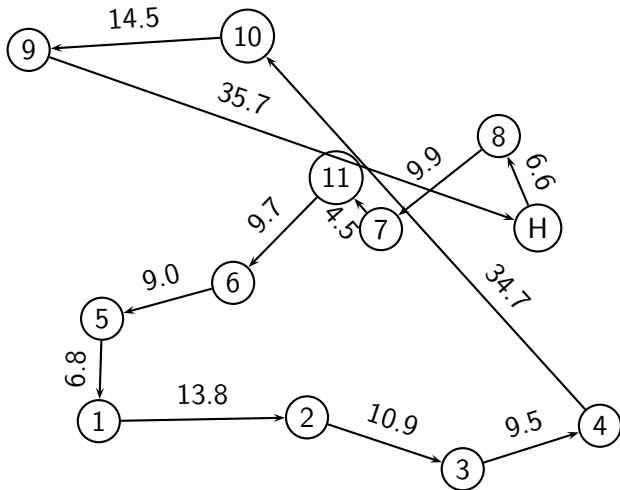


# Traveling salesman problem

## Example: 2-OPT(1,9)

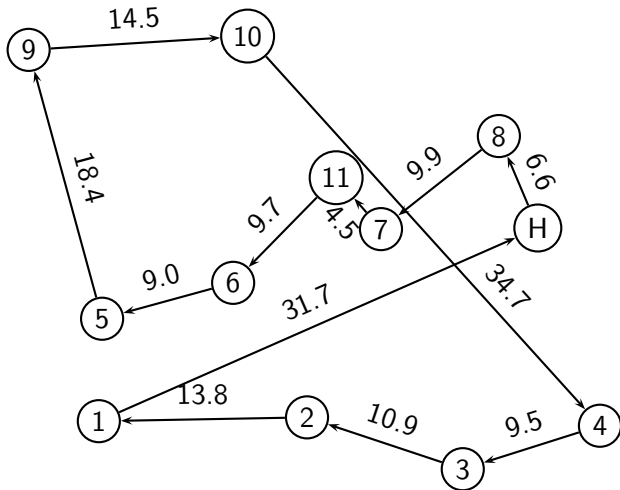
- Try to improve the solution using 2-OPT swapping 1 and 9.
- Before: H-8-7-11-6-5-1-2-3-4-10-9-H (length: 165.6)
- After : H-8-7-11-6-5-9-10-4-3-2-1-H (length: 173.3)
- No improvement.

# Neighborhood: 2-OPT(1,9)



Length: 165.6

# Neighborhood: 2-OPT(1,9)



Length: 173.3

# Exploration

## Comments

- Design of “vehicles” to explore the solution space.
- Fundamental neighborhoods exploit the structure of the problem.
- Various operations allow to modify and combine neighborhoods.
- Trade-off between flexibility and complexity.
- The neighborhood must be sufficiently large to increase the chances of improvement, and sufficiently small to avoid a lengthy enumeration.
- Example of a neighborhood too small: one neighbor at the west.
- Example of a neighborhood too large: each feasible point is in the neighborhood.

# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms
  - Brute force
  - Greedy heuristics
  - Exploration
  - **Intensification**
  - Diversification
- 4 Summary

## Local search: version one

- Consider the combinatorial optimization problem

$$\min f(x)$$

subject to

$$x \in \mathcal{U}.$$

- Consider the neighborhood structure  $V(x)$ , where  $V(x)$  is the set of feasible neighbors of  $x$ .
- At each iteration  $k$ , consider the neighbors in  $V(x_k)$  one at a time.
- For each  $y \in V(x_k)$ , if  $f(y) < f(x_k)$ , then  $x_{k+1} = y$  and proceed to the next iteration.
- If  $f(y) \geq f(x_k)$ ,  $\forall y \in V(x_k)$ ,  $x_k$  is a local minimum. Stop.

## Local search: version two

- Consider the combinatorial optimization problem

$$\min f(x)$$

subject to

$$x \in \mathcal{U}.$$

- Consider the neighborhood structure  $V(x)$ , where  $V(x)$  is the set of neighbors of  $x$ .
- At each iteration  $k$ , find  $y$  such that

$$f(y) \leq f(x_k), \forall y \in V(x_k).$$

- If  $f(y) = f(x_k)$ ,  $x_k$  is a local minimum. Stop.
- Otherwise, proceed to the next iteration.

# Local search: example

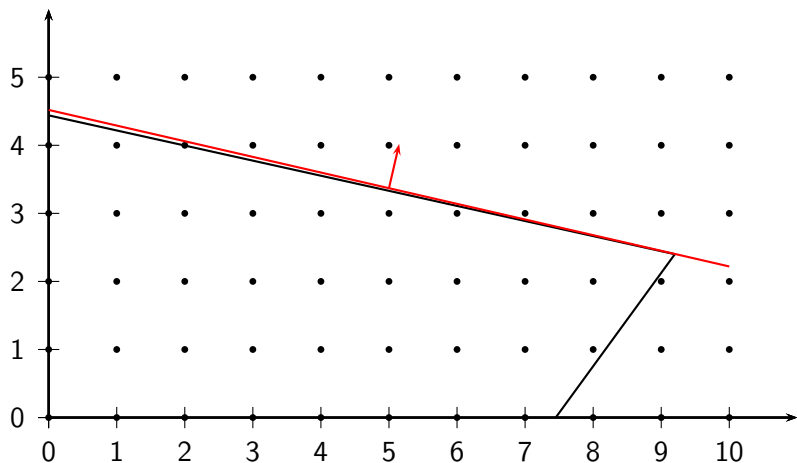
$$\min_{x \in \mathbb{R}^2} -3x_1 - 13x_2$$

subject to

$$\begin{aligned} 2x_1 + 9x_2 &\leq 40 \\ 11x_1 - 8x_2 &\leq 82 \\ x_1, x_2 &\in \mathbb{N} \end{aligned}$$

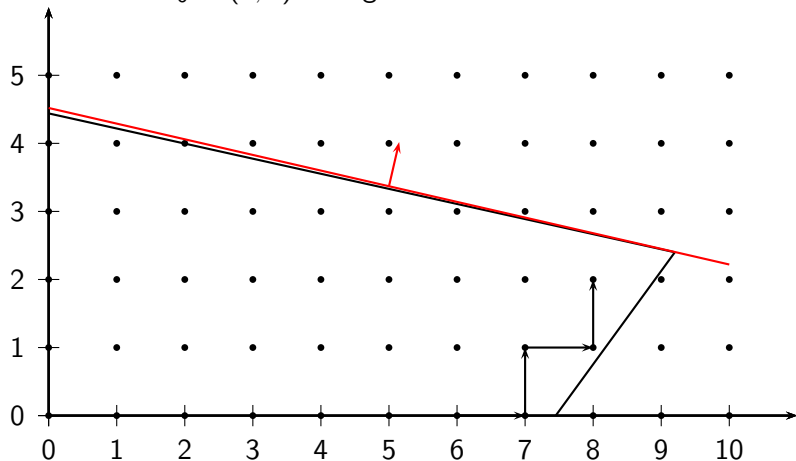


# Local search: example



# Local search: example

$x_0 = (6, 0)$  - Neighborhood: E - N - W - S



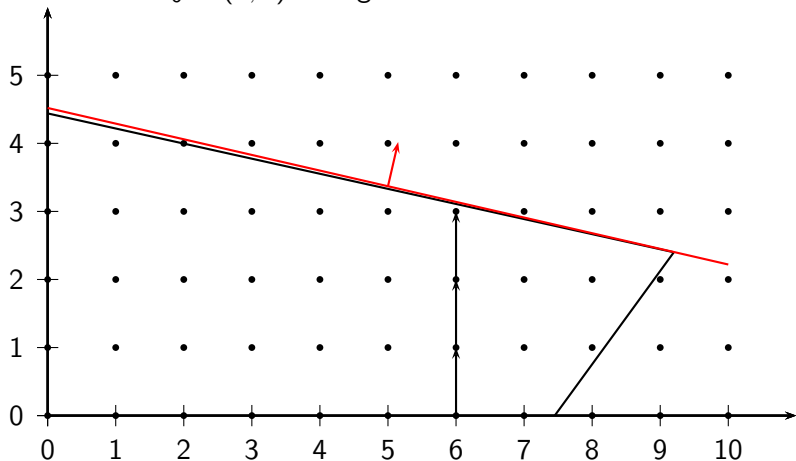
# Local search: example

$x_0 = (0, 3)$  - Neighborhood: E - N - W - S



# Local search: example

$x_0 = (6, 0)$  - Neighborhood : N - W - S - E



# The knapsack problem

$$\max_{x \in \{0,1\}^n} u^T x$$

subject to

$$w^T x \leq W.$$

# The knapsack problem

```
def localSearch(u, w, capacity, initSolution, neighborhood):
    x = initSolution
    ux = np.inner(u, x)
    wx = np.inner(w, x)
    if wx > capacity:
        Exception(f'Infeasible weight {wx} > {capacity}')
    localOptimum = False
    while not localOptimum:
        neighbors = neighborhood(x)
        localOptimum = True
        for y in neighbors:
            wy = np.inner(w, y)
            if wy <= capacity:
                uy = np.inner(u, y)
                if uy > ux:
                    localOptimum = False
                    x = y
                    ux = uy
                    wx = wy
```

# The knapsack problem

```
def neighborhood1(sack):
    return neighborhood(sack, size = 1, random = False, truncated = None)
firstSack = np.array([0]*n)

localSearch(utility, weight, capacity, firstSack, neighborhood1)
First sack: [0 0 0 0 0 0 0 0 0 0 0 0 0] U=0 W=0
New sack   : [1 0 0 0 0 0 0 0 0 0 0 0 0] U=80 W=84
New sack   : [0 0 0 0 0 1 0 0 0 0 0 0 0] U=84 W=96
New sack   : [1 0 0 0 0 1 0 0 0 0 0 0 0] U=164 W=180
New sack   : [1 1 0 0 0 1 0 0 0 0 0 0 0] U=195 W=207
New sack   : [1 0 1 0 0 1 0 0 0 0 0 0 0] U=212 W=227
New sack   : [1 0 0 0 0 1 0 0 0 1 0 0 0] U=222 W=233
New sack   : [1 0 0 0 0 1 0 0 0 0 0 1 0] U=231 W=258
New sack   : [1 1 0 0 0 1 0 0 0 0 0 1 0] U=262 W=285
New sack   : [1 0 0 0 0 1 1 0 0 0 0 1 0] U=265 W=300
```

# The knapsack problem

```
def neighborhood2(sack):
    return neighborhood(sack, size = 3, random = False, truncated = None)

firstSack = np.array([0]*n)
localSearch(utility, weight, capacity, firstSack, neighborhood2)
First sack: [0 0 0 0 0 0 0 0 0 0 0 0] U=0 W=0
New sack : [1 1 1 0 0 0 0 0 0 0 0 0] U=159 W=158
New sack : [1 1 0 0 0 1 0 0 0 0 0 0] U=195 W=207
New sack : [1 0 1 0 0 1 0 0 0 0 0 0] U=212 W=227
New sack : [1 0 0 0 0 1 0 0 0 1 0 0] U=222 W=233
New sack : [1 0 0 0 0 1 0 0 0 0 0 1] U=231 W=258
New sack : [0 1 0 0 0 1 0 0 0 1 0 1] U=240 W=254
New sack : [0 0 1 0 0 1 0 0 1 0 0 1] U=245 W=275
New sack : [0 0 1 0 0 1 0 0 0 1 0 1] U=257 W=274
New sack : [1 0 1 0 0 1 0 0 1 0 0 0] U=258 W=281
New sack : [1 0 1 0 0 1 0 0 0 1 0 0] U=270 W=280
New sack : [0 0 1 1 0 1 0 0 0 1 0 1] U=274 W=296
New sack : [0 0 1 0 1 1 0 0 0 1 0 1] U=284 W=295
New sack : [1 0 0 0 1 1 0 1 0 1 0 0] U=288 W=300
```

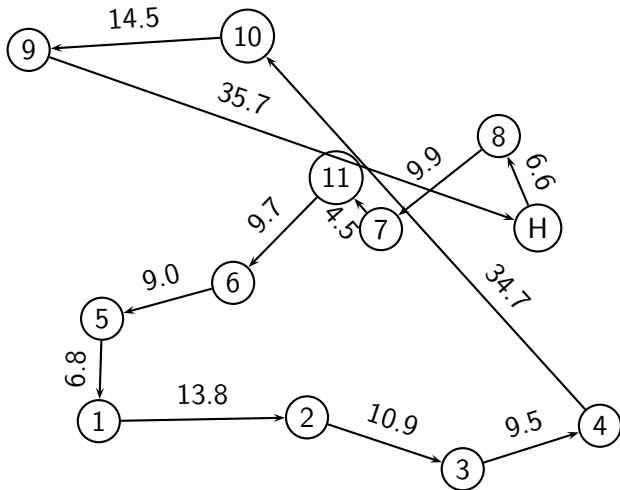


# Traveling salesman problem

## Procedure

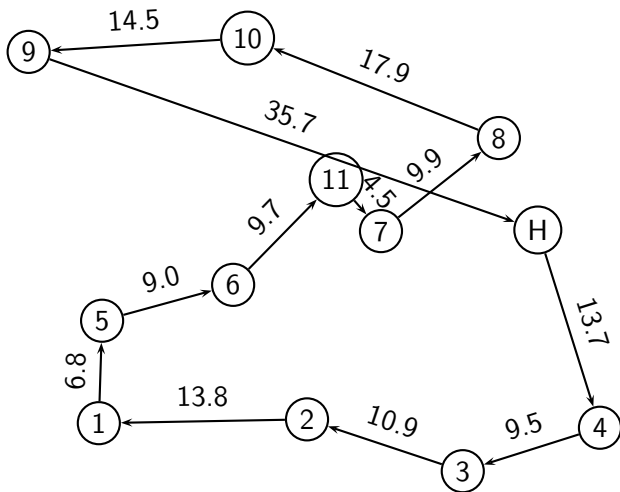
- Start with the outcome of the greedy algorithm.
- Use the 2-OPT neighborhood.
- Use version two of the local search.

## Current tour



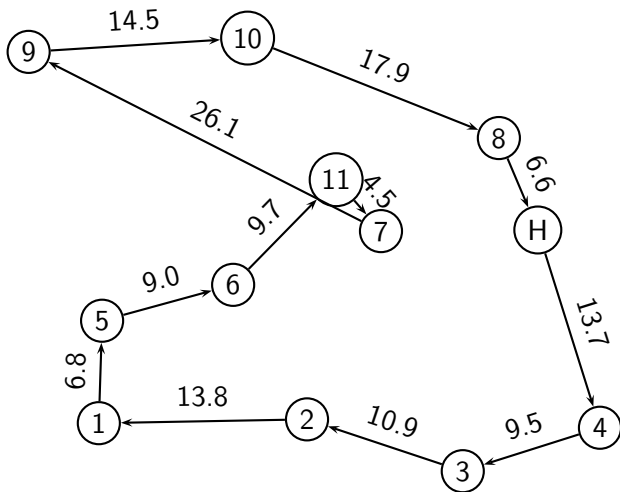
Length: 165.6

## Best neighbor: 2-OPT(8,4)



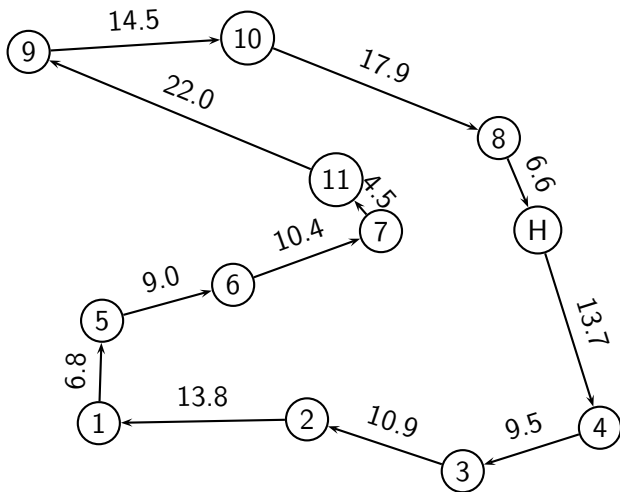
Length: 155.8

## Best neighbor: 2-OPT(8,9)



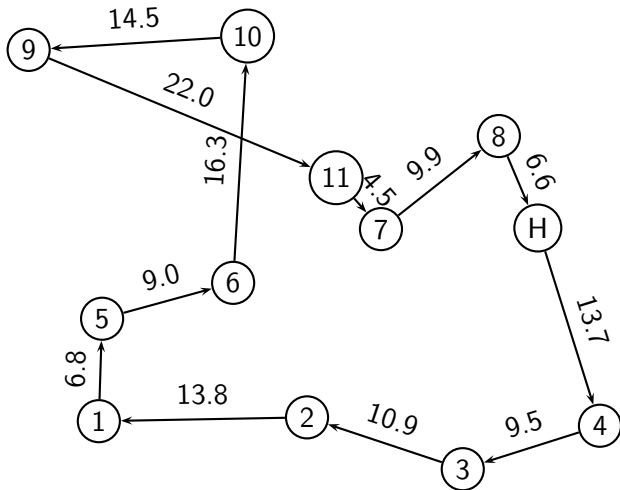
Length: 143.0

## Best neighbor: 2-OPT(11,7)



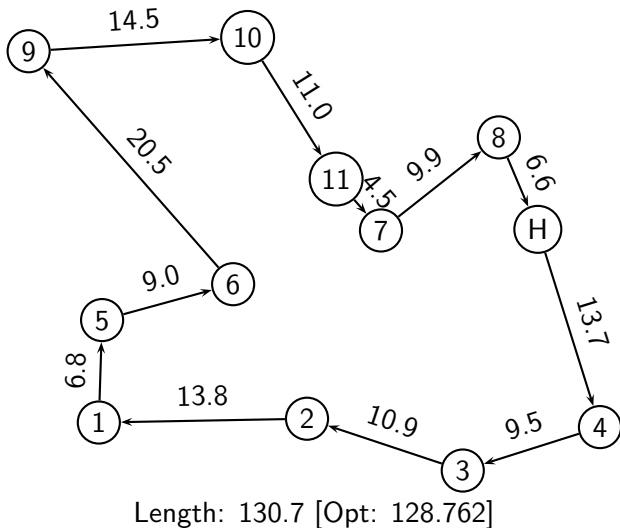
Length: 139.5

## Best neighbor: 2-OPT(7,10)



Length: 137.5

## Best neighbor: 2-OPT(10,9)



# Comments

- The algorithm stops at a local minimum, that is a solution better than all its neighbors.
- The outcome depends on the starting point and the structure of the neighborhood.
- Several variants are possible.



# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms
  - Brute force
  - Greedy heuristics
  - Exploration
  - Intensification
  - Diversification
- 4 Summary

# Changing the starting point

## Idea

- Launch the local search from several starting points.
- Select the best local optimum.

## Issues

- Feasibility.
- Same local optimum may be generated many times.
- Shooting in the dark.

# Variable Neighborhood Search

- aka VNS
- Idea: consider several neighborhood structures.
- When a local optimum has been found for a given neighborhood structure, continue with another structure.

# VNS: method

- Input**
- $V_1, V_2, \dots, V_K$  neighborhood structures.
  - Initial solution  $x_0$ .

- Initialization**
- $x_c \leftarrow x_0$
  - $k \leftarrow 1$

**Iterations** Repeat

- Apply local search from  $x_c$  using neighborhood  $V_k$

$$x^+ \leftarrow LS(x_c, V_k)$$

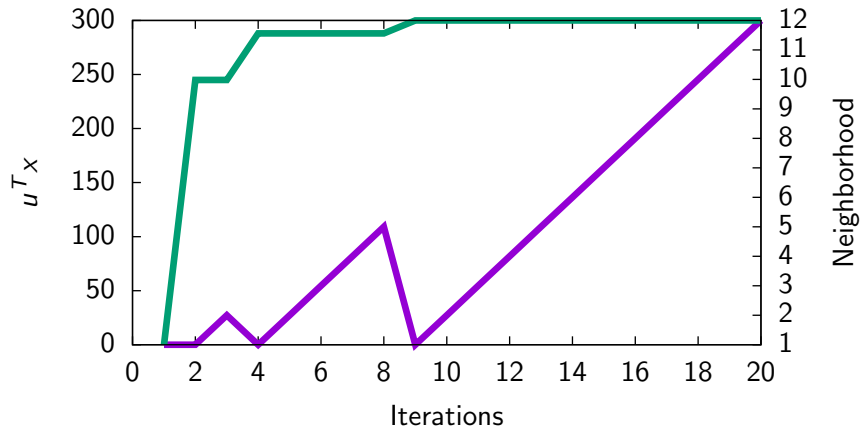
- If  $f(x^+) < f(x_c)$ , then  $x_c \leftarrow x^+$ ,  $k \leftarrow 1$ .
- Otherwise,  $k \leftarrow k + 1$ .

**Until**  $k = K$ .

# VNS: example for the knapsack problem

- Neighborhood of size  $k$ : modify  $k$  variables.
- Local search: current iterate:  $x_c$ 
  - randomly select a neighbor  $x^+$
  - if  $w^T x^+ \leq W$  and  $u^T x^+ > u^T x_c$ , then  $x_c \leftarrow x^+$

## VNS: example for the knapsack problem



# Simulated annealing

## Analogy with metallurgy

- Heating a metal and then cooling it down slowly improves its properties.
- The atoms take a more solid configuration.

## In optimization:

- Local search can both decrease and increase the objective function.
- At “high temperature”, it is common to increase.
- At “low temperature”, increasing happens rarely.
- Simulated annealing: slow cooling = slow reduction of the probability to increase.

# Simulated annealing

Modify the local search.

For the sake of simplicity: consider a neighborhood structure containing only feasible solutions.

Let  $x_k$  be the current iterate

- Select  $y \in V(x_k)$ .
- If  $f(y) \leq f(x_k)$ , then  $x_{k+1} = y$ .
- Otherwise,  $x_{k+1} = y$  with probability

$$e^{-\frac{f(y)-f(x_k)}{T}}$$

with  $T > 0$ .

Concretely, draw  $r$  between 0 and 1.

Accept  $y$  as next iterate if

$$e^{-\frac{f(y)-f(x_k)}{T}} > r$$



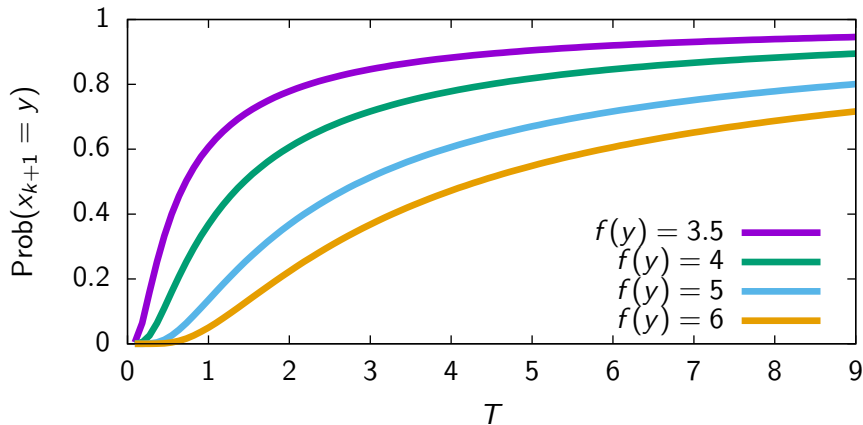
# Simulated annealing

$$\text{Prob}(x_{k+1} = y) = \begin{cases} 1 & \text{if } f(y) \leq f(x_k) \\ e^{-\frac{f(y)-f(x_k)}{T}} & \text{if } f(y) > f(x_k) \end{cases}$$

- If  $T$  is high (hot temperature), high probability to increase.
- If  $T$  is low, almost only decreases.

# Simulated annealing

Example :  $f(x_k) = 3$



# Simulated annealing

- In practice, start with high  $T$  for flexibility.
- Then, decrease  $T$  progressively.

# Simulated annealing

- Input**
- Initial solution  $x_0$
  - Initial temperature  $T_0$ , minimum temperature  $T_f$
  - Neighborhood structure  $V(x)$
  - Maximum number of iterations  $K$

**Initialize**  $x_c \leftarrow x_0, x^* \leftarrow x_0, T \leftarrow T_0$

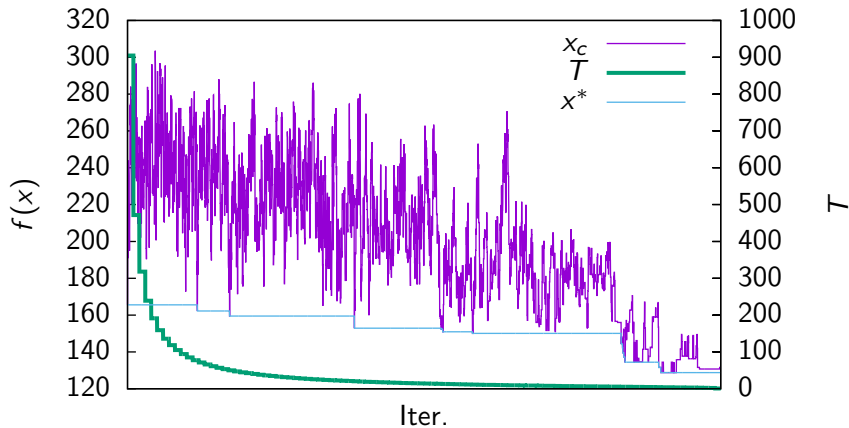
# Simulated annealing

Repeat  $k \leftarrow 1$

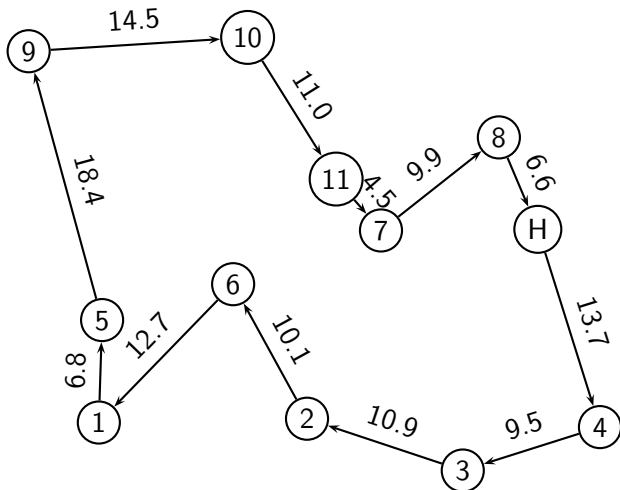
- While  $k < K$ 
  - Randomly select a neighbor  $y \in V(x_c)$
  - $\delta \leftarrow f(y) - f(x_c)$
  - If  $\delta < 0$ ,  $x_c = y$ .
  - Otherwise, draw  $r$  between 0 and 1
    - If  $r < \exp(-\delta/T)$ , then  $x_c = y$
    - If  $f(x_c) < f(x^*)$ ,  $x^* = x_c$ .
  - $k \leftarrow k + 1$
- Reduce  $T$

Until  $T \leq T_f$

# Example: traveling salesman problem



## Best solution found



Length : 128.8 [Optimal!]

## Practical comments

- Parameters must be tuned.
- In particular, the reduction rate of the temperature must be specified.
  - Let  $\delta_t$  be a typical increase of the objective function.
  - In the beginning, we want such an increase to be accepted with probability  $p_0$  (e.g.  $p_0 = 0.999$ )
  - At the end, we want such an increase to be accepted with probability  $p_f$  (e.g.  $p_f = 0.00001$ )
  - We allow for  $M$  updates of the temperature. So, for  $m = 0, \dots, M$ ,

$$T = - \frac{\delta_t}{\ln(p_0 + \frac{p_f - p_0}{M} m)}$$



# Comments

How to avoid being blocked in local minimum?

- Apply an algorithm from multiple starting points.
  - How to find feasible starting point?
  - How to avoid shooting in the dark?
- Change the structure of the neighborhood: **variable neighborhood search**
  - How to choose the neighborhood structures?
- Allow the algorithm to proceed upwards: **simulated annealing**
  - Climb the mountain to find another valley.
  - How to decide when it is time to climb or to go down?

# Outline

- 1 Motivation
- 2 Classical problems
- 3 Algorithms
  - Brute force
  - Greedy heuristics
  - Exploration
  - Intensification
  - Diversification
- 4 Summary

# Combinatorial optimization

## Characteristics

- $f$  and  $\mathcal{U}$  have no specific property.
- $f$  is a black box.
- $\mathcal{U}$  is a finite set of valid configurations.
- No optimality condition is available.

# Optimization methods

## Exact methods (branch and bound)

- Finds the optimal solution.
- Suffers from the curse of dimensionality.
- Requires the availability of valid and tight bounds.

## Approximation algorithms

- Finds a sub-optimal solution.
- Guarantees a bound on the quality of the solution.
- Mainly used for theoretical purposes.

## Heuristics

- Smart exploration of the solution space.
- No guarantee about optimality.
- Few assumptions about the problem.

# Heuristics: general framework

## Exploration

Neighborhood

## Intensification

Local search

## Diversification

Escape from local minima

# Meta-heuristics

- Methods designed to escape from local optima are sometimes called “meta-heuristics”.
- Plenty of variants are available in the literature.
- In general, success depends on exploiting well the properties of the problem at hand.
- VNS is one of the simplest to code.
- Additional bio-inspired methods have also been proposed and applied: genetic algorithms, ant colony optimization, etc.