

Heuristic Algorithms

Massimo Perego

Contents

Introduction	7
1 Possible CO Problems	11
1.1 Weighted set problems	11
1.1.1 The Knapsack Problem (KP)	11
1.2 Set problems in metric spaces	13
1.2.1 Maximum Diversity Problem (MDP)	13
Interlude 1: The objective function	14
1.3 Partitioning set problems	15
1.3.1 Bin Packing Problem (BPP)	15
1.4 Parallel Machine Scheduling Problem (PMSP)	17
Interlude 2: The objective function again	19
1.5 Logic function problems	20
1.5.1 Max-SAT (satisfaction) problem	20
1.6 Numerical matrix problems	22
1.6.1 Set Covering (SCP)	22
Interlude 3: The feasibility test	24
1.7 Numerical matrix problems	25
1.7.1 Set Packing	25
1.7.2 Set Partitioning (SPP)	27
Interlude 4: The search for feasible solutions	29
1.8 Graph problems	30
1.8.1 Vertex Cover (VCP)	30
1.8.2 Maximum Clique Problem	32
1.8.3 Maximum Independent Set Problem	34
Interlude 5: The relations between problems	36
1.8.4 Traveling Salesman Problem (TSP)	39
1.8.5 Capacitated Min. Spanning Tree Problem	41

1.8.6	Vehicle Routing Problem (VRP)	45
Interlude 6:	Combining alternative representations	47
2	Computational Complexity	48
2.1	Cost of an Algorithm	49
2.2	Worst-case asymptotic time complexity	50
2.2.1	The exhaustive algorithm	51
2.2.2	Polynomial and exponential complexity	51
2.3	Problem transformations and reductions	52
2.4	Beyond the worst-case complexity	53
2.4.1	Parameterized complexity	53
2.4.2	Kernelization (or “problem reduction”)	56
2.4.3	Average-case complexity	58
2.4.4	Phase transitions	60
	Computational cost of heuristic algorithms	62
3	Effectiveness of a heuristic algorithm	63
3.1	Distance	64
3.2	Theoretical analysis	65
3.2.1	Worst case	65
3.2.2	Beyond the worst case	74
3.2.3	Randomized approximation algorithms	75
3.3	Empirical analysis	77
3.3.1	Benchmark sample	79
3.3.2	Comparing heuristic algorithms	81
3.3.3	Analysis of the computational time	83
3.3.4	Analysis of the quality of the solution	87
3.3.5	Compact statistical descriptions	91
3.3.6	Boxplots diagrams	92
3.3.7	Relation between quality and computational time	94
3.4	Classification	95
3.4.1	Beyond the optimum, a generalization	96
3.4.2	Probability of success	97
3.5	Statistical tests (Wilcoxon’s test)	101
3.5.1	Assumptions	102
3.5.2	Application	103
4	Constructive Heuristics	106
4.1	Construction Graph	107
4.1.1	Termination condition	109

4.1.2	Definition of the construction graph	110
4.2	Effectiveness and Efficiency	112
4.3	General features	113
4.4	Examples	114
4.4.1	The Fractional Knapsack Problem (FKP)	114
4.4.2	The Knapsack Problem	115
4.4.3	Traveling Salesman Problem	116
4.4.4	Maximum Diversity Problem	117
4.4.5	Summary	118
4.4.6	Relevant Features	118
4.5	The additive case	119
4.5.1	Greedoids	120
4.5.2	Matroids	121
4.5.3	Greedoids with the strong exchange axiom	123
4.5.4	What to do when the axioms are violated	124
4.6	Heuristic constructive algorithms (HCA): the KP	127
4.6.1	2-Approximated algorithm for the KP	128
4.7	Pure and adaptive constructive algorithms	129
4.8	HCA: Set Covering	130
4.8.1	Approximability of the SCP	132
4.9	HCA: Bin Packing Problem	134
4.9.1	First-Fit heuristic	134
4.10	Extensions of the basic constructive scheme	137
4.10.1	Extensions of the construction graph	138
4.11	The Steiner Tree Problem (STP)	139
4.11.1	Distance Heuristic (DH) for the STP	140
4.12	Insertion algorithms for the TSP	143
4.12.1	Cheapest Insertion heuristic	144
4.12.2	Nearest Insertion heuristic	146
4.12.3	Farthest Insertion heuristic	148
4.13	Regret-based constructive heuristics	150
4.14	Roll-out heuristics	152
4.14.1	Generalized roll-out heuristics	154
4.15	Destructive heuristics	155
	Summary about constructive and destructive algorithms	158
5	Constructive Metaheuristics	159
5.1	Constructive metaheuristics	161
5.2	Adaptive Research Technique ART	162
5.2.1	Parameter tuning	165

5.2.2	Diversification and intensification	166
5.3	Semi-greedy heuristics	167
5.3.1	Convergence to the optimum	169
5.4	GRASP and Semi-greedy	171
5.4.1	Definition of the RCL	172
5.4.2	Reactive parameter tuning	174
5.5	Cost perturbation methods	175
5.6	Ant Colony Optimization	176
5.6.1	Trail	177
5.6.2	Random choice	178
5.6.3	Trail update	180
5.6.4	Convergence to the optimum	183
6	Exchange Algorithms	186
6.1	Neighborhoods	187
6.1.1	Neighborhoods based on distance	188
6.1.2	Neighborhoods based on operations	190
6.1.3	Distance and operation-based	191
6.1.4	Connectivity of the search graph	195
6.2	Steepest descent (hill-climbing) heuristics	196
6.2.1	Local and global optimality	197
6.2.2	Exact neighborhood	198
6.2.3	Properties of the search graph	199
6.3	Landscape	203
6.3.1	Autocorrelation coefficient	204
6.3.2	Plateau	206
6.3.3	Attraction basins	207
6.4	Complexity	208
6.4.1	The exploration of the neighborhood	208
6.4.2	Evaluating or updating the objective	210
6.5	Feasibility of the neighborhood	215
6.6	Partial saving of the neighborhood	217
6.7	Trade-off between efficiency and effectiveness	218
6.8	Very Large Scale Neighborhood Search	220
6.8.1	Efficient visit of exponential neighborhoods	221
6.8.2	Dynasearch	222
6.8.3	Cyclic exchanges	223
6.8.4	The improvement graph	224
6.9	Order-first split-second	229
6.10	Variable Depth Search (VDS)	231

6.10.1	Lin-Kernighan's algorithm for the symmetric TSP	234
6.11	Iterated greedy methods (destroy-and-repair)	236
6.12	Overcoming local optima	237
6.12.1	Termination condition	238
6.13	Modify the starting solution	239
6.13.1	Random generation	240
6.13.2	Constructive procedures	241
6.13.3	Influence of the starting solution	242
6.13.4	Exploiting the previous solutions	243
6.14	Iterated Local Search (ILS)	244
6.14.1	Perturbation procedure	246
6.14.2	Acceptance condition	247
6.15	Variable Neighborhood Search (VNS)	248
6.15.1	Adaptive perturbation mechanism	250
6.15.2	Skewed VNS	252
6.16	Extending the local search without worsening	253
6.17	Variable Neighborhood Descent (VND)	254
6.17.1	Neighborhood update strategies for the VND	256
6.18	Dynamic Local Search (DLS)	258
6.18.1	Variants	260
6.19	Extending the local search with worsenings	264
6.20	Simulated Annealing	265
6.20.1	Acceptance criteria	268
6.20.2	Asymptotic convergence to the optimum	269
6.20.3	Temperature update	271
6.21	Tabu Search	272
6.21.1	Reducing potential ineffectiveness	274
6.21.2	General scheme of the TS	277
6.21.3	Efficient evaluation of the tabu status	278
6.21.4	Tuning the tabu tenure	281
6.21.5	Variants	282
7	Recombination Metaheuristics	283
7.1	Scatter Search	285
7.1.1	Recombination procedure	287
7.2	Path Relinking	289
7.2.1	Variants	291
7.3	Encoding-based algorithms	292
7.4	Genetic Algorithm	293
7.4.1	Features of a good encoding	294

7.4.2	Encodings	296
7.4.3	Selection	300
7.4.4	Crossover	304
7.4.5	Mutation	306
7.4.6	The feasibility problem	307
7.5	Memetic algorithms	313
7.6	Evolution strategies	314

Introduction

The course aims to discuss the general aspects, the designing process of **Heuristic Algorithms**, along with methods to evaluate their performance.

An *algorithm* is a formal, deterministic procedure, with a correctness proof, while a **heuristic** is an informal, open rule, made from a bunch of common sense arguments.

An heuristic algorithm is an algorithm which does not guarantee a correct solution but can still be useful, provided that:

- it “costs” much less than a correct algorithm, in terms of time and space
- it frequently gets “close” to the solution; this requires a definition of “closeness” and a distribution to express the frequency of “good enough” solutions

Every algorithm always has a correctness proof while a heuristic is a “good idea” about the solution of a problem, that can become a proof if enlarged but doesn’t have to be, if you keep it a heuristic can provide a good result instead of a perfect one.

Restrictions from now on: heuristic algorithms

- that apply to **Combinatorial Optimization** problems
- that are **solution-based**

There are different type of problems, classified by the nature of the solution; the focus will be on a combination of **optimization** (looking for the optimal value) and **search** (looking for a subsystem assuming that value) problems.

An optimization/search problem can be represented as:

$$\text{opt } f(x), \quad x \in X$$

Where:

- a solution x describes each subsystem of the problem
- the feasible solution space X , the set of subsystems which satisfy given conditions
- the objective function $f : X \rightarrow \mathbb{R}$ quantitatively measures the quality of each subsystem ($\text{opt} \in \{\min, \max\}$)

The problem consists in determining:

- **optimization:** the optimal value f^* of the objective function:

$$f^* = \arg \underset{x \in X}{\text{opt}} f(x)$$

- **search:** at least one optimal solution, that is a subsystem:

$$x^* \in X^* = \arg \underset{x \in X}{\text{opt}} f(x) = \left\{ x^* \in X : f(x^*) = \underset{x \in X}{\text{opt}} f(x) \right\}$$

Exact optimization is costly and not always required, or even desirable; a heuristic is preferable.

Combinatorial Optimization: a problem is a *CO* problem when the feasible region X is a finite set, it has a finite number of feasible solutions.

This looks like a very restrictive assumption, however many problems can be reduced to finite set of solutions, e.g.:

- A problem can have a finite set of “interesting” solutions
- Some continuous problems can be reduced to CO problems (e.g. linear programming, Maximum Flow, ...)
- Continuous problems can be reduced to discrete ones by sampling (usually not very effective)
- Ideas conceived for CO problems can be extended to other problems (often quite effective)

A problem is a CO problem when:

1. the number of feasible solutions is finite
2. the feasible region is $X \subseteq 2^B$ for a given finite ground set B , that is, the feasible solutions are all subsets of the ground set that satisfy suitable conditions

The two definitions are equivalent. The latter allows a deeper analysis because X is not simply enumerated and X is defined in a compact way. The solution to a problem can be the subset of a finite set.

Classification of CO problems: solution-based heuristics consider solutions as subsets of the ground set:

- **constructive/destructive heuristics:** start from an extremely simple subset (\emptyset or B) and add/remove elements until the solution is obtained; the set only grows/shrinks
- **exchange heuristics:** start from a subset obtained in any way and exchange elements until the solution is found
- **recombination heuristics:** start from a population of subsets obtained in any way, recombine them producing a new population, taking parts from each or some of them

Heuristic designers can creatively combine elements from different classes.

There can be another distinction based on:

- the use of **randomization**:
 - purely deterministic heuristics
 - randomized heuristics, deterministic algorithms whose input includes pseudo-random numbers
- the use of **memory**: heuristics whose input
 - includes only the problem data
 - also includes previously generated solutions

These are independent of the previous classification.

Metaheuristics is the common name for heuristic algorithms with randomization and/or memory.

Risks to beware of when:

- **reverential or trendy attitude**, that is choosing an algorithm based on the social context, instead of the problem
- **magic attitude**, that is trusting a method on the basis of an analogy with physical and natural phenomena
- **heuristic integralism**, that is using a heuristic for a problem which admits exact algorithms
- **number crunching**, that is performing sophisticated and complex computations with unreliable numbers (that we don't understand)
- **SUV attitude**, that is relying on hardware power
- **overcomplication**, that is introducing redundant components and parameters, as if that could only improve the result
- **overfitting**, that is adapting components and parameters of the algorithm to the specific dataset used in the experimental evaluation

1 Possible CO Problems

First, a review of several problems to understand how to **apply abstract ideas**; the objective is to **find and exploit relations** between known and new problems.

The same idea can have different effectiveness on different problems.

1.1 Weighted set problems

1.1.1 The Knapsack Problem (KP)

Finding how much “stuff” you can put in the sack.

Selecting from a set of object a **subset of maximum value** which can be **contained in a knapsack** of limited capacity. It consists of:

- a set E of **elementary objects**
- a function $v : E \rightarrow \mathbb{N}$ describing the **volume of each object**
- a number $V \in \mathbb{N}$ describing the **capacity of the knapsack**
- a function $\phi : E \rightarrow \mathbb{N}$ describing the **value of each object**

The **ground set** is the set of **objects**, $B = E$.

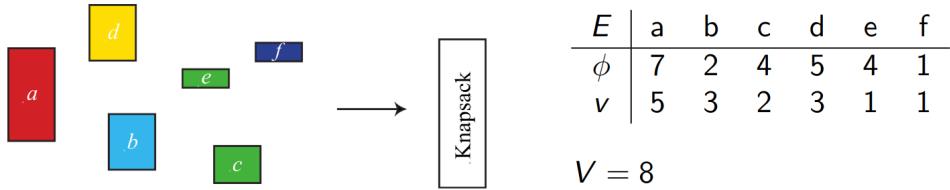
The **feasible region** includes all **subsets of object** whose total **volume does not exceed the capacity of the knapsack**

$$X = \left\{ x \subseteq B : \sum_{j \in x} v_j \leq V \right\}$$

The **objective** is to **maximize the total value** of the chosen objects

$$\max_{x \in X} f(x) = \sum_{j \in x} \phi_j$$

Example



$$x' = \{c, d, e\} \in X \quad f(x') = 13$$

$$x'' = \{a, c, d\} \notin X \quad f(x'') = 16$$

The left one is a solution, while the one on the right is not (or it can be called an unfeasible solution).

1.2 Set problems in metric spaces

1.2.1 Maximum Diversity Problem (MDP)

Select the k points with the maximum distance between each other.

Select **from a set of points** a **subset** of k points with the **maximum sum of all pairwise distances**. It consists of:

- a set P of **points**
- a function $d : P \times P \rightarrow \mathbb{N}$ providing the **distance between point pairs**
- a number $k \in \{1, \dots, |P|\}$ that is the **number of points** to select

The **ground set** is the set of **points** $B = P$.

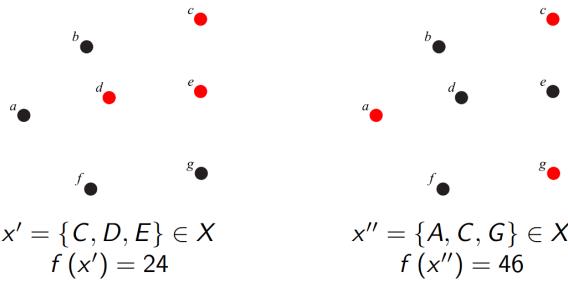
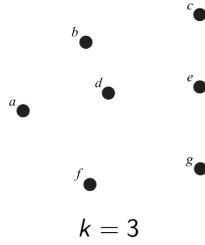
The **feasible region** includes all **subset** of k **points**

$$X = \{x \in B : |x| = k\}$$

The **objective** is to **maximize the sum** of all pairwise **distances** between the selected points

$$\max_{x \in X} f(x) = \sum_{(i,j): i,j \in x} d_{ij}$$

Example



Interlude 1: The objective function

The **objective function** associates integer values to feasible subjects

$$f : X \rightarrow \mathbb{N}$$

Computing the objective function can be complex (and exhausting).

In the cases seen before:

- the KP has an **additive** function which sums values of a function defined on the ground set (each element only has its own value)

$$\phi : B \rightarrow \mathbb{N} \text{ induces } f(x) \sum_{j \in x} \phi_j : X \rightarrow \mathbb{N}$$

the additive function is easy to recompute if the subset x changes, just sum each element added and subtract each element removed.

- the MDP has a **quadratic** objective function (each point needs to have the distance to every other element).
The process for recomputing, when the subset changes, is more complex, but still quadratic in nature.

Both of these functions are defined not only on X but on the whole 2^B (is it useful?).

1.3 Partitioning set problems

1.3.1 Bin Packing Problem (BPP)

Put a set of objects in the minimum possible number of containers of a given volume.

Divide a **set of voluminous objects** into the **minimum number of containers** of given capacity. It consists of:

- a set E of **elementary objects**
- a function $v : E \rightarrow \mathbb{N}$ describing the **volume of each object**
- a set C of **containers**
- a number $V \in \mathbb{N}$ that is the **volume of the containers**

The **ground set** includes all **(object, container) pairs**, $B = E \times C$.

The **feasible region** includes all **partitions** of the **objects** among the **containers** not exceeding the capacity of any container

$$X = \left\{ x \subseteq B : |x \cap B_e| = 1 \quad \forall e \in E, \sum_{(e,c) \in B^c} v_e \leq V \quad \forall c \in C \right\}$$

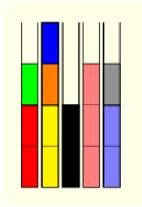
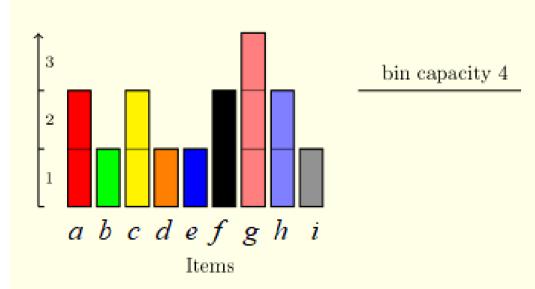
with $B_e = \{(i,j) \in B : i = e\}$ and $B^c = \{(i,j) \in B : j = c\}$ (respectively, pairs in relation to the elements and pairs in relation to the containers).

The **objective** is to **minimize the number of containers** used

$$\min_{x \in X} f(x) = |\{c \in C : x \cap B^c \neq \emptyset\}|$$

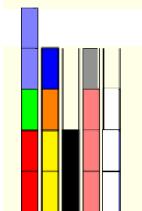
the cardinality of the set of containers C such that the pairs of the solution intersect B^c in at least one element (not empty).

Example



$$x' = \{(a, 1), (b, 1), (c, 2), (d, 2), (e, 2), (f, 3), (g, 4), (h, 5), (i, 5)\} \in X$$

$$f(x') = 5$$



$$x'' = \{(a, 1), (b, 1), (c, 2), (d, 2), (e, 2), (f, 3), (g, 4), (h, 1), (i, 4)\} \notin X$$

$$f(x'') = 4$$

In the first solution there is only a single pair for each item and the capacity of each container is not exceeded so it's a valid solution.

1.4 Parallel Machine Scheduling Problem (PMSP)

Divide a **set of tasks** among a **set of machines minimizing the completion time**. It consists of:

- a set T of **tasks**
- a function $d : T \rightarrow \mathbb{N}$ describing the **time length of each task**
- a set M of **machines**

divide the tasks among the machines with the minimum completion time, each task to one machine only.

The **ground set** includes all **(task,machine) pairs**, $B = T \times M$.

The **feasible region** includes all **partitions of tasks among machines** (the order of the tasks is irrelevant, sum is the same)

$$X = \{x \subseteq B : |x \cap B_t| = 1 \quad \forall t \in T\}$$

The **objective** is to **minimize the maximum sum of time lengths** for each machine

$$\min_{x \in X} f(x) = \max_{m \in M} \sum_{t:(t,m) \in x} d_t$$

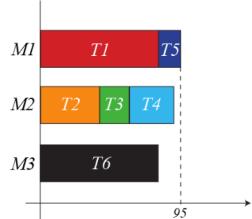
this wants to minimize the maximum sum of duration for each machine in the solution; the final “value” (time spent) needs to be at a minimum and it is determined only by the longest-running machine.

Example

$$T = \{T_1, T_2, T_3, T_4, T_5, T_6\}$$

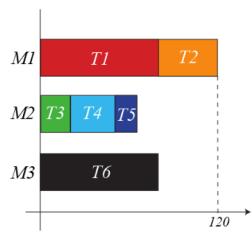
$$M = \{M_1, M_2, M_3\}$$

task	T_1	T_2	T_3	T_4	T_5	T_6
d	80	40	20	30	15	80



$$x' = \{(T_1, M_1), (T_2, M_2), (T_3, M_2), (T_4, M_2), (T_5, M_1), (T_6, M_3)\} \in X$$

$$f(x') = 95$$



$$x'' = \{(T_1, M_1), (T_2, M_1), (T_3, M_2), (T_4, M_2), (T_5, M_2), (T_6, M_3)\} \in X$$

$$f(x'') = 120$$

The objective function wants to minimize the completion time, the value to complete the whole set of tasks, i.e. it wants to minimize the maximum time of a single machine.

Interlude 2: The objective function again

The **ground set** is **not** always a single set, in the last examples was a cartesian product of two sets.

The objective function for the BPP and PMSP is not additive and not trivial to compute.

Small **changes** in the solution have a **variable impact** on the objective, it can be:

- **equal** to the time length of the moved task (e.g., move T_5 on M_1 in x'')
- **zero** (e.g., move T_5 on M_3 in x'')
- **intermediate** (e.g., move T_2 on M_2 in x'')

The **impact** of a change to the solution **depends** both on the **modified** elements and the **unmodified** ones (contrary to the earlier interlude).

The objective function is “flat”, there are **several solutions** with the same **value** (is there a way to tell if a change is “good”?).

1.5 Logic function problems

1.5.1 Max-SAT (satisfaction) problem

Given a *CNF* (Conjunctive Normal Form), assign truth values to its logical variables so as to satisfy the **maximum weight subset of its logical clauses**. It consists of:

- a set V of **logical variables** x_j with values in $\mathbb{B} = \{0, 1\}$ (*false, true*);
the variables taken into consideration, which can be true or false
- a **literal** l_j is a function consisting of an **affirmed or negated variable**

$$l_j(x) \in \{x_j, \bar{x}_j\}$$

- a **logical clause** is a disjunction or **logical sum (OR)** of literals

$$C_i(x) = l_{i,1} \vee \dots \vee l_{i,n}$$

- a **conjunctive normal form (CNF)** is a conjunction or **logical product (AND)** of logical formulae

$$CNF(x) = C_1 \wedge \dots \wedge C_n$$

it's the (logic) product of (logic) sums (literals).

- to satisfy a logical function means to make it assume value 1
- a function w provides the **weights** of the *CNF* formulae

The **ground set** is the set of **all simple truth assignments**

$$B = V \times \mathbb{B} = \{(x_1, 0), (x_1, 1), \dots, (x_n, 0), (x_n, 1)\}$$

the cartesian product of the set of variables with the set of logical values.

The **feasible region** includes all **subsets of simple assignments** that are:

- **complete**, every variable has at least one value
- **consistent**, every variable has at most one value

$$X = \{x \subseteq B : |x \cap B_v| = 1 \forall v \in V\}$$

with $B_{x_j} = \{(x_j, 0), (x_j, 1)\}$.

The **objective** is to **maximize the total weight of the satisfied formulae**:

$$\max_{x \in X} f(x) = \sum_{i: C_i(x)=1} w_i$$

This is defined only on feasible solutions (you HAVE to assign a value to a variable, it just might not be the Max-SAT solution).

Example

- Variables

$$V = \{x_1, x_2, x_3, x_4\}$$

- Literals

$$L = x_1, \bar{x}_1, x_2, \bar{x}_2, x_3, \bar{x}_3, x_4, \bar{x}_4$$

- Logical clauses

$$C_1 = \bar{x}_1 \vee x_2 \quad \dots \quad C_7 = x_2$$

- Conjunctive normal form

$$CNF = (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_4) \wedge x_1 \wedge x_2$$

- Weight function (uniform, all 1):

$$w_i = 1 \quad i = 1, \dots, 7$$

$x = \{(x_1, 0), (x_2, 0), (x_3, 1), (x_4, 1)\}$ satisfies $f(x) = 5$ formulae out of 7.
Complementing a variable does not always change $f(x)$ (x_1 does, x_4 doesn't).

1.6 Numerical matrix problems

1.6.1 Set Covering (SCP)

Given

- a **binary matrix** $A \in \mathbb{B}^{m,n}$ with **row set** R and **column set** C
- column $j \in C$ **covers** row $i \in R$ when $a_{ij} = 1$
- a function $c : C \rightarrow \mathbb{N}$ provides the **cost of each column**

Select a **subset of columns covering all rows at minimum cost**.

Essentially: there's a binary matrix (filled with 0s and 1s), each column has a cost, you need to get the minimum cost for covering all the rows, i.e. having at least one "1" value for each row.

The **ground set** is the set of **columns**, $B = C$.

The **feasible region** includes all **subsets** of **columns** that **cover all rows**

$$X \left\{ x \subseteq B : \sum_{j \in x} a_{ij} \geq 1 \forall i \in R \right\}$$

All the solution subset of the columns such that there is at least a 1 ($a_{ij} \geq 1$) for all rows i .

The **objective** is to **minimize** the total **cost** of the selected **columns**

$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

It's additive and is simply the cost of the columns.

Example

c	4	6	10	14	5	6
A	0	1	1	1	1	0
	0	0	1	1	0	0
	1	1	0	0	0	1
	0	0	0	1	1	1
	1	1	1	0	1	0
A	0	1	1	1	1	0
	0	0	1	1	0	0
	1	1	0	0	0	1
	0	0	0	1	1	1
	1	1	1	0	1	0
A	0	1	1	1	1	0
	0	0	1	1	0	0
	1	1	0	0	0	1
	0	0	0	1	1	1
	1	1	1	0	1	0

$$x' = \{c_1, c_3, c_5\} \in X$$

$$f(x') = 19$$

$$x'' = \{c_1, c_5, c_6\} \notin X$$

$$f(x'') = 15$$

The first one is a solution, the second one is unfeasible.

“Set Covering”: covering a set (rows) with subsets (columns).

Interlude 3: The feasibility test

Often heuristic algorithms require solving the problem “here is a subset of elements, **is it feasible?**” or in short “ $x \in X?$ ”. It’s a decision problem.

Sometimes is really easy (Max-SAT problem, just check that each variable appears exactly once), sometimes it’s not as linear (SCP, you need the sum of each row, still easy but not as easy).

The **feasibility test** requires to **compute from the solution** and test

- **a single number:** the total volume (KP), the cardinality (MDP)
- **a single set of numbers:** values assigned to each variable (Max-SAT), number of machines for each task (PMSP)
- **several sets of numbers:** number of containers for each object and total volume of each container (BPP)

The time required can be **different** if the test is performed

- **from scratch** on a generic subset x
- on a **subset** x' obtained by slightly **modifying** a feasible solution x

Some modifications can be forbidden *a priori* to avoid infeasibility (insertions and removals for MDP, PMSP, Max-SAT), while others require an *a posteriori* test (exchanges). Some families of modification can be ruled out.

1.7 Numerical matrix problems

1.7.1 Set Packing

Given:

- a **binary matrix** $A \in \mathbb{B}^{m,n}$ with **row set** R and **column set** C
- **columns** j' e $j'' \in C$ **conflict** with each other when $a_{ij'} = a_{ij''} = 1$
- a function $\phi : C \rightarrow \mathbb{N}$ provides the **value of each column**

Select a **subset** of **nonconflicting columns** of **maximum value**.

There can't be two 1s in the same row of two chosen columns; the sum of all values in each row must be ≤ 1 .

The **ground set** is the set of **columns**, $B = C$.

The **feasible region** includes all **subsets** of **nonconflicting columns**

$$X = \left\{ x \subseteq B : \sum_{j \in x} a_{ij} \leq 1 \forall i \in R \right\}$$

The **objective** is to **maximize the total value** of the selected columns

$$\max_{x \in X} f(x) = \sum_{j \in x} \phi_j$$

Example

	ϕ	<table style="border-collapse: collapse; width: 100%;"> <tr><td style="width: 15px;"></td><td style="width: 15px;">4</td><td style="width: 15px;">6</td><td style="width: 15px;">10</td><td style="width: 15px;">14</td><td style="width: 15px;">5</td><td style="width: 15px;">6</td></tr> </table>		4	6	10	14	5	6																													
	4	6	10	14	5	6																																
A		<table style="border-collapse: collapse; width: 100%;"> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td></tr> </table>		0	1	0	0	1	0		0	0	1	1	0	0		1	0	0	0	0	1		0	0	0	1	1	1		1	1	1	0	0	0	
	0	1	0	0	1	0																																
	0	0	1	1	0	0																																
	1	0	0	0	0	1																																
	0	0	0	1	1	1																																
	1	1	1	0	0	0																																
A		<table style="border-collapse: collapse; width: 100%;"> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td></tr> </table>		0	1	0	0	1	0		0	0	1	1	0	0		1	0	0	0	0	1		0	0	0	1	1	1		1	1	1	0	0	0	$x' = \{c_2, c_4\} \in X$ $f(x') = 20$
	0	1	0	0	1	0																																
	0	0	1	1	0	0																																
	1	0	0	0	0	1																																
	0	0	0	1	1	1																																
	1	1	1	0	0	0																																
A		<table style="border-collapse: collapse; width: 100%;"> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">1</td><td style="width: 15px;">2</td></tr> <tr><td style="width: 15px;"></td><td style="width: 15px;">1</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">0</td><td style="width: 15px;">1</td><td style="width: 15px;">2</td></tr> </table>		0	1	0	0	1	0		0	0	1	1	0	0		1	0	0	0	0	1		0	0	0	1	1	2		1	0	0	0	1	2	$x'' = \{c_1, c_5, c_6\} \notin X$ $f(x'') = 15$
	0	1	0	0	1	0																																
	0	0	1	1	0	0																																
	1	0	0	0	0	1																																
	0	0	0	1	1	2																																
	1	0	0	0	1	2																																

The first one is a feasible solution, while the second one is not feasible (the sum two of the rows is > 1).

“Set Packing”: packing disjoint subsets (columns) of a set (rows).

1.7.2 Set Partitioning (SPP)

Given a binary matrix and a cost function defined on its columns, select a **minimum cost** subset of **nonconflicting columns covering all rows**. It consists of:

- a **binary matrix** $A \in \mathbb{B}^{m,n}$ with a **set of rows** R and a **set of columns** C
- a function $c : C \rightarrow \mathbb{N}$ that provides the **cost of each column**

similar to the last problems, the difference is that the sum in each row must be exactly = 1; cover each row exactly once, not at least once (SCP) or at most once (Set Packing).

The **ground set** is the set of columns, $B = C$.

The **feasible region** includes all subsets of **columns** that **cover all rows** and are **not conflicting**

$$X = \left\{ x \subseteq C : \sum_{j \in x} a_{ij} = 1 \forall i \in R \right\}$$

The **objective** is to **minimize** the total **cost** of the selected **columns**

$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Example

c	4	6	10	14	5	6
A	0	1	0	0	1	0
	0	0	1	1	1	0
	1	0	0	0	0	1
	0	0	0	1	1	0
	1	1	1	0	0	0
A	0	1	0	0	1	0
	0	0	1	1	0	1
	1	0	0	0	0	1
	0	0	0	1	1	0
	1	1	1	0	0	0

$$x' = \{c_2, c_4, c_6\} \in X$$

$$f(x') = 26$$

$$x'' = \{c_1, c_5, c_6\} \notin X$$

$$f(x'') = 15$$

The first one is a feasible solution, while the second one is not feasible (the sum of a row is 0, while another one is 2).

“Set Partitioning”: partition a set (rows) into subsets (columns).

Interlude 4: The search for feasible solutions

Heuristic algorithms often require to solve the problem “**Find a feasible solution $x \in X$** ”, it’s a search problem.

Depending on the problem, the solution can be **trivial**:

- some sets are **always feasible**, such as $x = \emptyset$ (KP, Set Packing) or $x = B$ (feasible instances of SCP)
- random solutions satisfying a **constraint**, such as $|x| = k$ (MDP, doesn’t matter which points, you can take k random points)
- random solutions satisfying **consistency constraints**, such as assigning one task to each machine (PMSP), one value to each logic variable (Max-SAT), etc.; a random assignment (in both cases) can be a feasible solution, it might be a bad solution, but feasible

but it can also be **hard**:

- in the BPP the number of containers must be sufficiently large (e.g., provide one container for each object, then minimize)
- in the SPP no polynomial algorithm is known to solve the problem

Some algorithms **enlarge the feasible region** from X to X' (relaxation)

- the **objective f** must be **extended** from X to X' (sometimes it’s possible, sometimes it’s not, see first interlude)
- but often $X' \setminus X$ includes better solutions; the new solutions could be better (obviously, we are relaxing the constraints, the objective function can be better for unfeasible solutions)

1.8 Graph problems

1.8.1 Vertex Cover (VCP)

Given an **undirected graph**, select a **subset of vertices of minimum cardinality** such that **each edge of the graph is incident to it**. It consists of:

- an **undirected graph** $G = (V, E)$

The goal is to get every node adjacent to at least one selected node.

The problem can be weighted by adding a weight function to each node instead of considering unitary value.

The **ground set** is the **vertex set**, $B = V$.

The **feasible region** includes all vertex subsets such that **all the edges of the graph are incident to them**

$$X = \{x \subseteq V : x \cap (i, j) \neq \emptyset \forall (i, j) \in E\}$$

all the subsets of vertices such as the intersection between this and every possible pair of edges is not empty, i.e. there must be every pair of vertices in the solution.

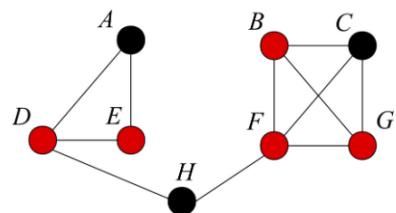
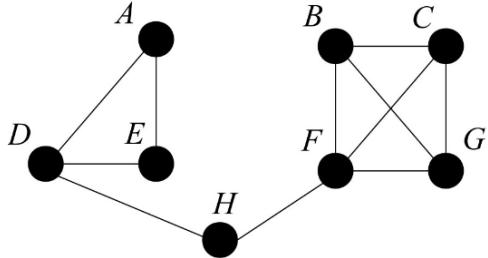
The **objective** is to **minimize** the **number of selected vertices**

$$\min_{x \in X} f(x) = |x|$$

It becomes the sum of the weight of the vertices if we have a weight function w

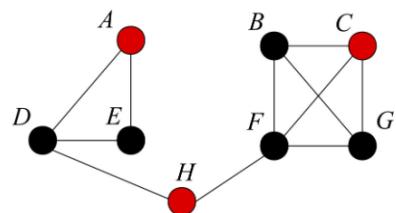
$$\min_{x \in X} f(x) = \sum_{j \in x} w_j$$

Example



$$x' = \{B, D, E, F, G\} \in X$$

$$f(x') = 5$$



$$x'' = \{A, C, H\} \notin X$$

$$f(x'') = 3$$

The first one is a solution, the second one is unfeasible since there are some edges missing.

As the last interlude predicted, the unfeasible solution has a better value than the feasible one.

1.8.2 Maximum Clique Problem

Given:

- an **undirected graph** $G = (V, E)$
- a function $w : V \rightarrow \mathbb{N}$ that provides the weight of each vertex

select the subset of **pairwise adjacent vertices of maximum weight**.

The problem consists in finding the subset of adjacent vertices (a clique, each vertex is connected by an edge to every other node) of maximum weight.

The **ground set** is the **vertex set**, $B = V$.

The **feasible region** includes all **subsets of pairwise adjacent vertices**

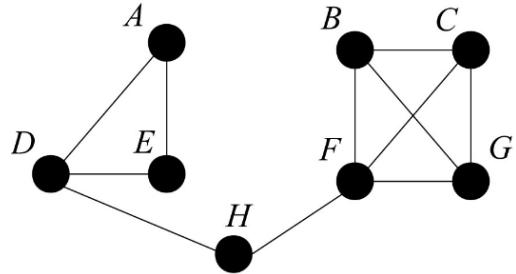
$$X = \{x \subseteq V : (i, j) \in E \ \forall i \in x, \forall j \in X \setminus \{i\}\}$$

for any pair of vertices belonging to the solution the pair itself belongs to the edge set.

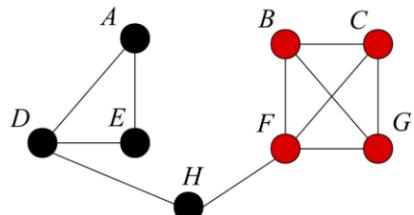
The **objective** is to **maximize** the **weight** of the selected vertices

$$\max_{x \in X} f(x) = \sum_{j \in x} w_j$$

Example

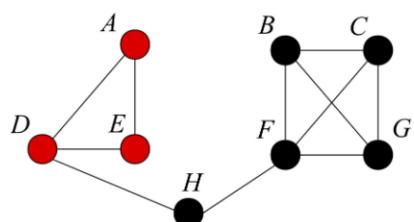


Uniform weights: $w_i = 1$ for each $i \in V$



$$x' = \{B, C, F, G\} \in X$$

$$f(x') = 4$$



$$x'' = \{A, D, E\} \in X$$

$$f(x'') = 3$$

1.8.3 Maximum Independent Set Problem

Given

- an **undirected graph** $G = (V, E)$
- a function $w : V \rightarrow \mathbb{N}$ that provides the **weight of each vertex**

select the **subset of pairwise nonadjacent vertices of maximum weight**.

The subset of not connected nodes with the maximum weight.

The **ground set** is the **vertex set**, $B = V$.

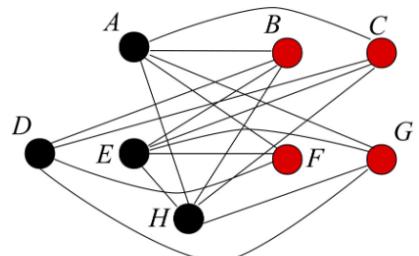
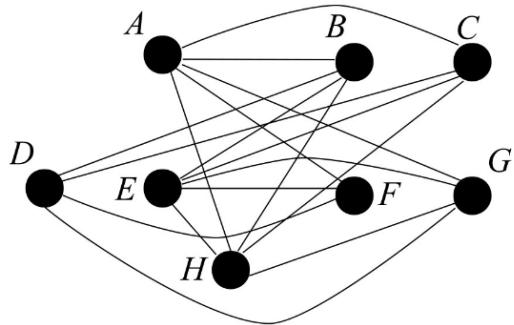
The **feasible region** includes the subsets of **pairwise nonadjacent vertices**

$$X = \{x \subseteq B : (i, j) \notin E \ \forall i \in x, \forall j \in x \setminus \{i\}\}$$

The **objective** is to **maximize the weight** of the selected vertices

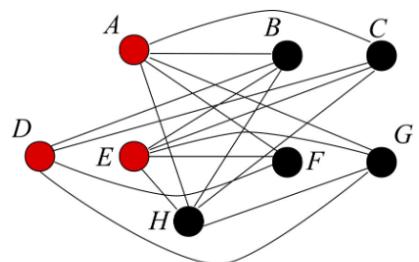
$$\max_{x \in X} f(x) = \sum_{j \in x} w_j$$

Example



$$x' = \{B, C, F, G\} \in X$$

$$f(x') = 4$$



$$x'' = \{A, D, E\} \in X$$

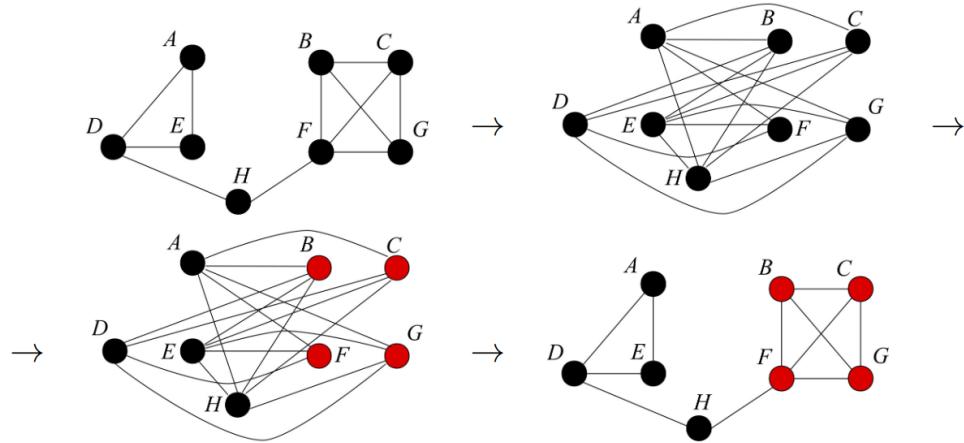
$$f(x'') = 3$$

—

Interlude 5: The relations between problems

Each instance of the **MCP** is **equivalent** to an instance of the **MISP**:

1. start from the MCP instance, that is graph $G = (V, E)$
2. build the complementary graph $\bar{G} = (V, (V \times V) \setminus E)$
3. find an optimal solution of the MISP on \bar{G}
4. the corresponding vertices give an optimal solution of the MCP on G
(a heuristic MISP solution gives a heuristic MCP solution)



The process can also be applied in the **opposite** direction.

The **solution of the MCP can be used to find the solution of the MISP**, and vice versa.

There's no need to design two different algorithms since the same one works for both problems, with a simple transformation of the instance.

The **VCP** and the **SCP** are also **related**, but in a different way; each instance of the VCP can be **transformed** into an **instance** of the SCP:

- each edge i corresponds to a row of the covering matrix A
- each vertex j corresponds to a column of A
- if edge i touches vertex j , set $a_{ij} = 1$; otherwise $a_{ij} = 0$
- an optimal solution of the SCP gives an optimal solution of the VCP
(a heuristic SCP solution gives a heuristic VCP solution)

	A	B	C	D	E	F	G	H
(A, D)	1	0	0	1	0	0	0	0
(A, E)	1	0	0	0	1	0	0	0
(B, C)	0	1	1	0	0	0	0	0
(B, F)	0	1	0	0	0	1	0	0
(B, G)	0	1	0	0	0	0	1	0
(C, F)	0	0	1	0	0	1	0	0
(C, G)	0	0	1	0	0	0	1	0
(D, E)	0	0	0	1	1	0	0	0
(D, H)	0	0	0	1	0	0	0	1
(F, G)	0	0	0	0	0	1	1	0
(F, H)	0	0	0	0	0	1	0	1

The reverse is not as simple, not every binary matrix can be turned into a graph.

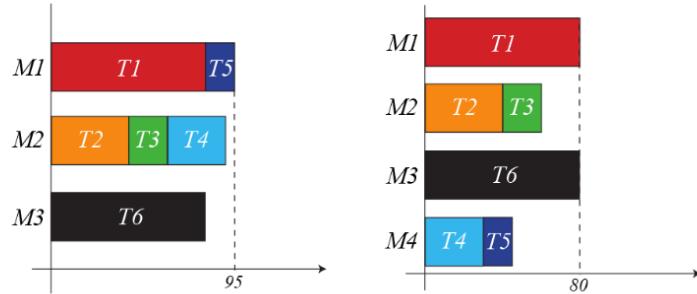
The VCP is equivalent to solving the SCP on a matrix that determines to which vertices every edge is connected to.

The **BPP** and the **PMSP** are **equivalent**, but in a more sophisticated way:

- the **tasks** correspond to the **objects**
- the **machines** correspond to the **containers**, but
 - BPP: minimise the number of containers, given the capacity
 - PMSP: given the number of machines, minimise the completion time

Start from a **BPP instance**

1. make an **assumption** on the optimal **number of containers** (e.g., 3)
 2. build the **corresponding PMSP instance**
 3. **compute the optimal completion time** (e.g., 95)
 - if it **exceeds** the capacity (e.g., 80), increase the assumption (4 or 5)
 - if it **does not**, decrease the assumption (2 or 1)
- (using heuristic PMSP solutions leads to a heuristic BPP solution)



The reverse process is *possible*.

The two problems are equivalent, but each one must be solved several times.

1.8.4 Traveling Salesman Problem (TSP)

Given:

- a **directed graph** $G = (N, A)$
- a function $c : A \rightarrow \mathbb{N}$ that provides the **cost of each arc**

select a **circuit** visiting **all the nodes** of the graph at **minimum cost**.

The **ground set** is the **arc set**, $B = A$.

The **feasible region** includes the **circuits** that **visit all nodes** in the graph (hamiltonian circuits).

How to determine whether a subset is a feasible solution?

And a modification of a feasible solution?

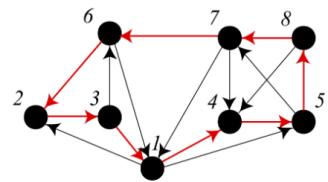
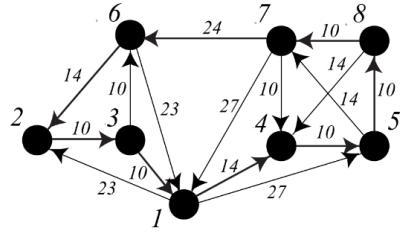
Can some modifications be ruled out?

Finding an hamiltonian circuit is usually NP-hard, it's trivial only in the case of a connected graph.

The **objective** is to **minimize** the **total cost** of the selected arcs

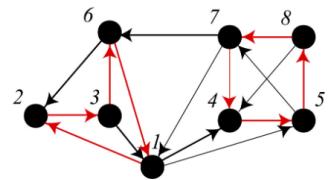
$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Example



$$x' = \{(1, 4), (4, 5), (5, 8), (8, 7), (7, 6), (6, 2), (2, 3), (3, 1)\} \in X$$

$$f(x') = 102$$



$$x'' = \{(4, 5), (5, 8), (8, 7), (7, 4), (1, 2), (2, 3), (3, 6), (6, 1)\} \notin X$$

$$f(x'') = 106$$

The first one is a solution, the second one isn't because although it visits all the vertices there are 2 different sub-circuits.

1.8.5 Capacitated Min. Spanning Tree Problem

Given

- an **undirected graph** $G = (V, E)$ with a **root vertex** $r \in V$
- a function $c : E \rightarrow \mathbb{N}$ that provides the **cost of each edge**
- a function $w : V \rightarrow \mathbb{N}$ that provides the **weight of each vertex**
- a number $W \in \mathbb{N}$ that is the **subtree appended to the root** (branch), i.e. capacity of each subtree

select a **spanning tree** of **minimum cost** such that each **branch respects the capacity**.

Find a spanning tree starting from a root node r , each branch must be under the limit weight (not too many vertices, must be under W).

The **ground set** is the **edge set**, $B = E$.

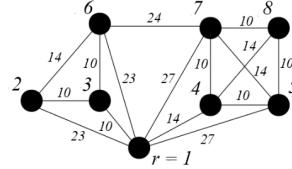
The **feasible region** includes all **spanning trees** such that the **weight of the vertices** spanned by each branch **does not exceed W** .

The feasibility test requires to visit the subgraph.

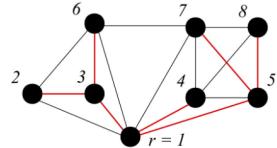
The **objective** is to **minimize** the **total cost** of the selected edges

$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Example

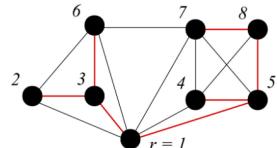


Uniform weight ($w_i = 1$ for each $i \in V$) and capacity: $W = 3$



$$x' = \{(r, 3), (3, 2), (3, 6), (r, 4), (r, 5), (5, 7), (5, 8)\} \in X$$

$$f(x') = 95$$



$$x'' = \{(r, 3), (3, 2), (3, 6), (r, 5), (5, 4), (5, 8), (8, 7)\} \notin X$$

$$f(x'') = 87$$

It is easy to evaluate the objective, less easy the feasibility.

Cost of the main operation: the objective function is:

- **fast to evaluate:** sum the edge cost
- **fast to update:** sum the added costs and subtract the removed ones

but it's easy to generate nonoptimal subtrees given the covered vertices.

The **feasibility test** is

- **not very fast to perform:**
 - visit to check for connection and acyclicity
 - visit to compute the total weight of each subtree
- **not very fast to update:**
 - show that the removed edges break the loops introduced by the added ones
 - recompute the weights of the subtrees

This also holds when the graph is complete.

What if we described the problem in terms of vertex subsets?

Alternative description: define a set of **subtrees** T (as in the containers in the BPP), one for each vertex in $V \setminus \{r\}$: some can be empty.

The **ground set** is the **set of the (vertex,branch) pairs**, $B = V \times T$.

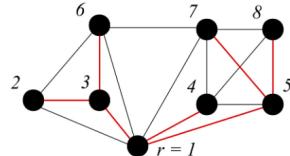
The **feasible region** includes **all partitions of the vertices into connected subsets** (visit, trivial on complete graphs) **of weight $\leq W$** (as in the BPP)

$$X = \left\{ x \subseteq B : |x \cap B_v| = 1 \forall v \in V \setminus \{r\}, \sum_{(i,j) \in B^t} w_i \leq W \forall t \in T, \dots \right\}$$

with $B_v = (i, j) \in B : i = v, B^t = (i, j) \in B : j = t$.

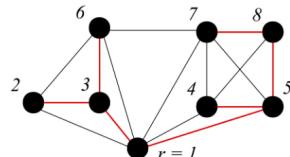
The **objective** is to **minimize the sum of the costs of the branches** spanning each subset of vertices and appending it to the root.
It is a combination of minimum spanning tree problems.

The previously considered solutions now have a **different representation**.



$$x' = \{(2, T1), (3, T1), (6, T1), (4, T2), (5, T3), (7, T3), (8, T3)\} \in X$$

$$f(x') = 95$$



$$x'' = \{(2, T1), (3, T1), (6, T1), (4, T2), (5, T2), (7, T2), (8, T2)\} \notin X$$

$$f(x'') = 87$$

The feasibility test only requires to sum the weights, computing the objective requires to solve a MST problem.

New cost of operations: the objective function is

- **slow to evaluate:** compute a MST for each subset
- **slow to update:** recompute the MST for each modified subset

but the subtrees are **optimal by construction.**

If the graph is complete, the **feasibility test** is

- **fast to perform:**
 - sum the weights of the vertices for each subtree
- **fast to update:**
 - sum the added weights and subtract the removed ones

There are advantages and disadvantages switched places.

1.8.6 Vehicle Routing Problem (VRP)

Given

- a **directed graph** $G = (N, A)$ with a **depot node** $d \in N$
- a function $c : A \rightarrow \mathbb{N}$ that provides the **cost of each arc**
- a function $w : N \rightarrow \mathbb{N}$ that provides the **weight of each node**
- a number $W \in \mathbb{N}$ that is the **capacity of each circuit**

select a **set of circuits of minimum cost** such that **each one visits the depot and respects the capacity**.

It's a set of circuits visiting all the nodes, if the TSP had more salesmen and each salesman has a maximum capacity of stuff he can bring.

The **ground set** could be

- the **arc set**, $B = A$
- the **set of all (node,circuit) pairs**, $B = N \times C$

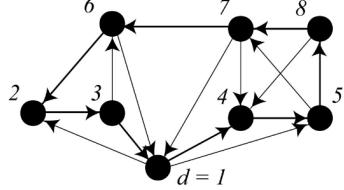
The **feasible region** could include

- all **arc subsets** that **cover all nodes** with circuits **visiting the depot** and whose **weight does not exceed W** (again the visit of a graph)
- all **partitions** of the nodes into **subsets of weight non larger than W** and **admitting a spanning circuit** (NP-hard problem)

The **objective** is to **minimize the total cost** of the selected arcs

$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Example



Uniform weight ($w_i = 1$ for each $i \in N$) and capacity: $W = 4$

The solutions could be described as

- arc subsets
 $x = \{(d, 2), (2, 3), (3, 6), (6, d), (d, 4), (4, 5), (5, 8), (8, 7), (7, d)\} \in X$
 - node partitions
 $x = \{(2, C1), (3, C1), (6, C1), (4, C2), (5, C2), (7, C2), (8, C2)\} \in X$
- $$f(x) = 137$$

Interlude 6: Combining alternative representations

The **CMSTP** and the **VRP** share an interesting complication: **different definitions of the ground set B** are possible and natural

- the description as a **set of edges/arcs** looks preferable to **manage the objective**
- the description as a **set of pairs (vertex,tree)/(node,circuit)** looks better to **generate optimal solutions and to deal with feasibility**

Which description should be adopted?

- the one that makes easier the most frequent operations
- both, if they are used much more frequently than updated, so that the burden of keeping them up-to-date and consistent is acceptable

2 Computational Complexity

Problems: informally, a problem is a **question** on a **system** of **mathematical objects**. The same question can often be asked on many similar systems

- an **instance** $i \in I$ is each specific system concerned by the question
- a **solution** $s \in S$ is an answer corresponding to one of the instances

Formally, a problem is the **function which relates instances and solutions**

$$P : I \rightarrow S$$

Defining a function doesn't mean knowing how to compute it.

Algorithms: an algorithm is a **formal procedure**, composed by **elementary steps**, in **finite sequence**, each determined by an input and by the results of the previous steps.

An algorithm for a **problem** P is an algorithm which, given in **input** $i \in I$, returns in **output** $S_i \in S$

$$A : \rightarrow S$$

An algorithm **defines a function** plus the **way to compute** it; it can be

- **exact** if its associated function coincides with the problem
- **heuristic** otherwise

A heuristic algorithm is **useful** if it is

1. **efficient**: it "costs" much less than an exact algorithm
2. **effective**: it "frequently" provides a solution "close" to the right one

2.1 Cost of an Algorithm

The “cost” of an algorithm denotes the **computational cost** of running it:

- **space** occupied in memory
- **time** required to terminate

The **time** is usually considered more important, since:

- space is **renewable**, can be reused, time isn’t
- using **space** always **requires time**
- usage of space is usually **easier** to **distribute** (among machines)

Space and time are **partly interchangeable**, it’s possible to reduce use of one by increasing the other.

Physical time is dependent on too many unreliable factors, and the measure of **computational time** should be

- **unrelated to technology**, i.e. the same for different machines
- **concise**, summarized in a simple symbolic expression
- **ordinal**, sufficient to compare different algorithms

2.2 Worst-case asymptotic time complexity

The Worst-case asymptotic time complexity provides such a measure through the passages of:

1. define the **time** T as the **number of elementary operations** performed (independent from the specific machine)
2. define the **size of an instance** as a suitable value n (e.g. the number of elements in the ground set)
3. find the **worst-case**, the maximum of T on all instances of size n

$$T(n) = \max_{i \in I_n} T(i) \quad n \in \mathbb{N}$$

Now time complexity is a function $T : \mathbb{N} \rightarrow \mathbb{N}$.

4. **approximate** $T(n)$ from above/below with a simpler function $f(n)$, considering only their **asymptotic behavior** (for $n \rightarrow +\infty$, we want an efficient algorithm on instances of large sizes)
5. collect the function in **classes** with the **same approximating functions**

Θ functional spaces:

$$T(n) \in \Theta(f(n))$$

Represents the **tight bound** of a function's growth rate, $T(n)$ must be "enclosed" between $c_1 f(n)$ and $c_2 f(n)$, from a "large" value of n , i.e. it's valid from a point n_0 forward.

Asymptotically, $f(n)$ **estimates** $T(n)$ up to a multiplying factor.

O functional spaces:

$$T(n) \in O(f(n))$$

Represents the **upper bound** of a function's growth rate, $c \cdot f(n)$ "dominates" $T(n)$, from a point n_0 forward.

Asymptotically, $f(n)$ **overestimates** $T(n)$ up to a multiplying factor.

Ω functional spaces:

$$T(n) \in \Omega(f(n))$$

Represents the **lower bound** of a function's growth rate, $T(n)$ dominates $\Omega(n)$ from a point n_0 forward.

Asymptotically, $f(n)$ **underestimates** $T(n)$ up to a multiplying factor.

2.2.1 The exhaustive algorithm

Within CO problems. Define the **size** of an instance as the **cardinality** of the **ground set**

$$n = |B|$$

The exhaustive algorithm

- considers **each** and every **subset** $x \subseteq B$, that is each $x \in 2^{|B|}$
- test its **feasibility** ($x \in X$) in time $\alpha(n)$
- if positive, **evaluates** the **objective** $f(x)$ in time $\beta(n)$
- **updates** the best value found so far, if needed

The **time complexity** becomes

$$T(n) \in \Theta(2^n (\alpha(n) + \beta(n)))$$

This is **at least exponential**, even if $\alpha(n)$ and $\beta(n)$ are small polynomials (which is often the case).

The exhaustive algorithm is impractical most of the time.

2.2.2 Polynomial and exponential complexity

In CO, the main **distinction** is between:

- **polynomial complexity**: $T(n) \in O(n^d)$ for a constant $d > 0$
- **exponential complexity**: $T(n) \in \Omega(d^n)$ for a constant $d > 1$

The **first** family includes **efficient** algorithms, the **second inefficient** ones.

In general, the **heuristic** algorithms are **polynomial** algorithms for problems whose known **exact** algorithms are all **exponential**.

2.3 Problem transformations and reductions

Problems can be **transformed** into other problems (Interlude 5). A **relation** among problems allows to design algorithms:

- by **transformation**:
 1. given I_P (instance of P) **build** I_Q
 2. given I_Q , **apply** A_Q to obtain S_Q (solution of I_Q)
 3. given S_Q , **build** S_P
- by **reduction**: **repeat** the transformation, correcting I_Q based on the **solutions obtained**; iterate the process

If A_Q is exact/heuristic, A_P is exact/heuristic.

The two algorithms often have a **similar complexity**, if A_Q is polynomial/exponential and

- building I_Q takes polynomial time
- the number of iterations is polynomial
- building S_P takes polynomial time

then A_P is polynomial/exponential.

2.4 Beyond the worst-case complexity

The worst-case complexity **cancels** all information on the **easier instances** (maybe i just need to use easy instances) and gives a rough **overestimate** of the computational time, in some (rare) case useless.

What if the **hard** instances are **rare** in practice? To **compensate** one can investigate:

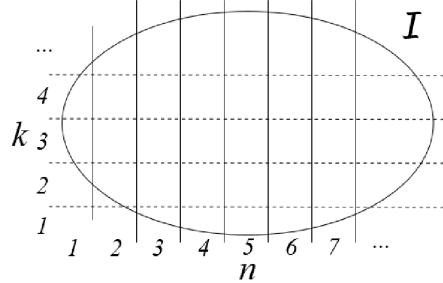
- **parameterized complexity**, that is introduce some **other relevant parameter** k (besides the size n) and express the **time** as $T(n, k)$
- **average-case complexity**, that is assume a **probability distribution** on I and express the **time** as the **expected value**

$$T(n) = E [T(i)|i \in I_n]$$

2.4.1 Parameterized complexity

Some algorithms are **exponential** in k an **polynomial** in n and thus:

- **efficient** on instances with **low** k
- **inefficient** on instances with **large** k



Nature of k : it could be part of the **input**:

- a **numerical constant** (e.g. the capacity of the KP)
- the **maximum number of literals** per formula in logic function problems
- the **number on nonzero elements** in numerical matrix problems
- the **maximum degree, the diameter**, etc... in graph problems

one knows *a priori* whether the algorithm is efficient on a given instance.

If the additional parameter k is part of the **solution**:

- the **cardinality** of the solution

one will only find out *a posteriori* (but an estimate *a priori* could be available).

Example: the VCP; Exhaustive algorithm: for each of the 2^n subsets of vertices, test if it covers all edges, compute its cardinality and keep the smallest one

$$T(n, m) \in \Theta(2^n(m + n))$$

If we already know that there's a solution with $f(x) = |x| = k + 1$, we can look for a solution of k vertices, decreasing k progressively; if we can't find it we already have the optimal solution.

Naive algorithm: for each subset of k vertices, test if it covers all edges

$$T(n, m, k) \in \Theta(n^k m)$$

for a fixed k , this algorithm is polynomial (but in general very slow).

Bounded tree search: a better algorithm can be based on the following useful property

$$x \cap (u, v) \neq \emptyset \text{ for all } x \in X, (u, v) \in E$$

Any feasible solution **includes** at least one **extreme vertex** for each edge. Choose any edge (there can be “good” and “bad” edges), only one must be in the solution.

BTS algorithm to find x with $|x| \leq k$:

1. choose **any** (u, v) : either $u \in x$ or $u \notin x$ and $v \in x$ (there must be only one in the solution)
2. for each open case, **remove** the **vertices** of x and the **edges they cover**

$$V := V \setminus x \quad E := E \setminus \{e \in E : e \cap x \neq \emptyset\}$$

The edges covered by vertices in x are no longer constraining

3. if $|x| \leq k$ and $E = \emptyset$, x is the **required solution**; if the partial solution has k elements (or less) and there are no more edges to cover then it's not a partial solution, it's a feasible one
4. if $|x| \leq k$ and $E \neq \emptyset$, there is **no solution**
5. otherwise go to step 1

The **complexity** is $T(n, m, k) \in \Theta(2^k m)$, polynomial in n ($m < n^2$).

For $n \gg 2$, this algorithm is much more efficient than the naive one.

2.4.2 Kernelization (or “problem reduction”)

Kernelization **transforms** all **instances** of P into **simpler** instances of P (also known as “problem reduction”), instead of instances of another problem Q .

Quite often, in fact, useful **properties** allow to **prove that**

- there exists an optimal solution (not every optimal solution, but at least one) **not including** certain elements of $B \Rightarrow$ such elements can be **removed**
- there exists an optimal solution **including** certain elements of $B \Rightarrow$ such elements can be set **apart and added later**

In short, remove elements of B without affecting the solution.

Possible useful **outcomes** are

- an **exact** algorithm **polynomial** in n (the problem becomes simpler, parameterized complexity)
- **faster** exact and heuristic algorithms (the instance is smaller so it's faster and usually better since the smaller problem is less “confusing”)
- **better** heuristic **solutions**
- **heuristic kernelization:** apply **relaxed conditions** sacrificing optimality (risk losing the optimal solution but simplify the problem)

Kernelization of VCP: if $\delta_v \geq k + 1$ (the degree of v , number of vertices), vertex v belongs to any feasible solution of value $\leq k$ (you need to cover a lot of edges, this covers a lot of edges; by contradiction, you can't cover $k + 1$ edges with less than this single vertex).

Kernelization algorithm to **keep** only **vertices of solution** x with $|x| \leq k$

- start at **step** $t = 0$ with $k_0 = k$ and an **empty** vertex **subset** $x_t := \emptyset$
- **set** $t = t + 1$ and **add** to the solution the **vertices of degree** $\geq k_t + 1$

$$\delta_v \geq k + 1 \implies x_t := x_{t-1} \cup \{v\}$$

- **update** $k_t := k_0 - |x_t|$
- **remove** the **vertices of zero degree**, those **of** x and the **covered edges**

$$V := \{v \in V : \delta_v > 0\} \setminus x_t \quad E := \{e \in E : e \cap x_t = \emptyset\}$$

- if $|E| > k_t^2$ there is **no feasible solution** (k_t vertices are not enough)
- if $|E| \leq k_t^2 \implies |V| \leq 2k_t^2$; **apply** the **exhaustive** algorithm

The **complexity** is $T(n, k) \in \Theta(n + m + 2^{2k^2} k^2)$ (the last part is the one of the exhaustive algorithm, good if k is small).

Essentially: it chooses the nodes with the most edges connected to them, removes them (and the nodes left with no edges), decreases the number of edges “required” to be “chosen” and starts again.

2.4.3 Average-case complexity

Some algorithms are inefficient only on a few instances (see simplex algorithm for Linear Programming).

Idea: define time as the **expected value** of $T(i)$ on I_n for each $n \in \mathbb{N}$

$$T(n) = E[T(i)|i \in I_n]$$

Theoretical studies can define a **probabilistic model** of the problem, that is a probability **distribution** on I_n for each $n \in \mathbb{N}$ (probability of each instance, typically quite simple, could be equiprobability) and **compute** the **expected value** of $T(I)$.

Empirical studies

- build a **simulation model** of the problem, that is a probability distribution on I_n for each $n \in \mathbb{N}$, let it be theoretical or empirical (drawn from real-world data)
- build a **benchmark** of **random** instances according to the **distribution**
- apply the algorithm and **measure** the **time** required

Probabilistic models for numerical matrices: Binary random matrix with a given size (m rows and n columns), some models are:

1. **equiprobability:** list all 2^{mn} binary matrices and select one of the matrices with uniform probability
2. **uniform probability:** set each cell to 1 with a given probability p

$$Pr[a_{ij} = 1] = p \quad (i = 1, \dots, m; j = 1, \dots, n)$$

If $p = 0.5$, it coincides with the equiprobability model, for other values some instances are more likely than others

3. **fixed density:** extract δmn (obviously $0 < \delta < 1$) cells out of mn with uniform probability and set them to 1.
If $\delta = p$, it resembles the uniform probability model, but some instances cannot be generated

Probabilistic models for graph: Random graph with a given number of vertices n , some models are:

1. **equiprobability:** list all $2^{\frac{n(n-1)}{2}}$ graphs and select one of the graphs with uniform probability
2. **Gilbert's model, or uniform probability $G(n, p)$:**

$$Pr [(i, j) \in E] = p \quad (i \in V, j \in V \setminus \{i\})$$

All graphs with the same number of edges m have the same probability $p^m(1-p)^{\frac{n(n-1)}{2}-m}$ (different for each m).

If $p = 0.5$, it coincides with the equiprobability model

3. **Erdős-Rényi model $G(n, m)$:** extract m unordered vertex pairs out of $\binom{n}{2}$ with uniform probability and create an edge for each one
If $\frac{2m}{n(n-1)} = p$, it resembles the uniform probability model, but some instances cannot be generated

Probabilistic models for logic functions: Random CNF with a given number of variables n and a given number of literals k for each logic formula

1. **fixed-probability ensemble:** list all $\binom{n}{k} 2^k$ formulae of k distinct and consistent literals and add each one to the CNF with probability p
2. **fixed-size ensemble:** build m formulae, adding to each one k distinct and consistent literals, extracted with uniform probability.
If $p = \frac{m}{\binom{n}{k} 2^k}$, it resembles the fixed-probability model, but some instances cannot be generated

2.4.4 Phase transitions

Different values of the (deterministic or probabilistic) **parameters** correspond to **different regions** of the **instance** set.

If you introduce parameters, different regions of the instance set are obtained.

For **graphs**

- $m = 0$ and $p = 0$ correspond to **empty graphs**
- $m = \frac{n(n-1)}{2}$ and $p = 1$ correspond to **complete graphs**
- intermediate values correspond to graphs of **intermediate density** (deterministically for m , probabilistically for p)

You can expect the algorithm to perform differently on different regions.

For many problems the **performance** of algorithms is strongly **different in different regions** concerning

- the **computational time** (for exact and heuristic algorithms)
- the **quality of the solution** (for heuristic algorithms)

Often, the performance **variation** takes place **abruptly in small regions** of the parameter space, as in phase transitions of physical systems (e.g. ice to water happens in a restricted zone).

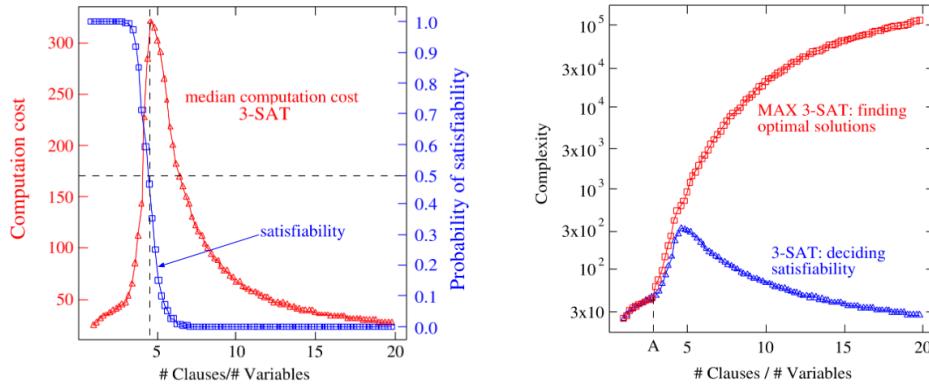
This is useful to predict the behavior of an algorithm on a given instance.

Phase transitions for 3-SAT and Max-3-SAT: Given a CNF on n variables, with logic formulae containing 3 literals

- **3-SAT:** is there a truth assignment satisfying all formulae?
- **Max-3-SAT:** what is the maximum number of satisfiable formulae?

As the formulae/variables ratio, $\alpha = m/n$, increases

- **satisfiable instances decrease** from nearly all (many variables for few formulae) to nearly none (few variables for many formulae)
- the **computing time** first **sharply increases**, then **decreases** for SAT, increases further for Max-SAT (using a well-known exact algorithm)

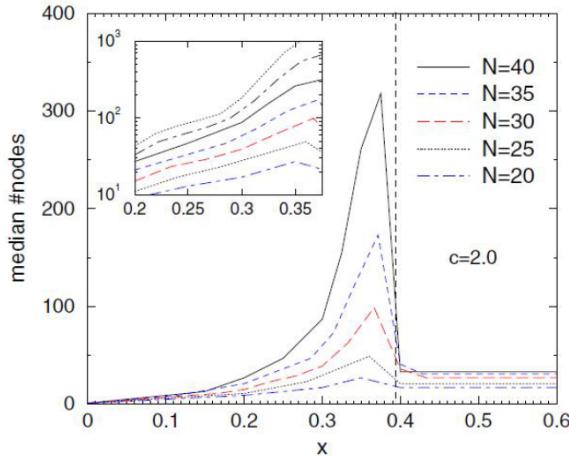


The blue profile in the first graph (3-SAT) shows the number of instances that can be satisfied for a certain number of clauses/variables. The red shows the computational time (on a specific algorithm, but many have the same basic ideas), which starts small, gets big and then decreases (as stated before), there's a range, where the lines meet, which makes it hard to prove the satisfiability of the clauses.

In second graph (Max-3-SAT) the blue profile shows the complexity of deciding satisfiability (same as red in the first graph, re-scaled), the red one shows the complexity of finding optimal solutions, which only increases.

As $n \rightarrow +\infty$, the transition **concentrates** around $\alpha_c \approx 4.26$.

Phase transition for the VCP: The VCP exhibits a **similar phase transition** as $|x|/|V|$ increases, the computational time first explodes, then drops, as $n \rightarrow +\infty$ **concentrates** around a critical value.



When $|x|/|V|$ is small some vertices are clearly necessary, when it's large many vertices are clearly necessary; in both cases, problem solved.

Computational cost of heuristic algorithms

The **time complexity** of a heuristic algorithm is usually

- **strictly polynomial** (with low exponents)
- fairly **robust** with respect to **secondary parameters**

Therefore, the worst-case estimation is also good on average.

Metaheuristics use random steps or memory

- the **complexity** is well defined for **single components** of the algorithm
- the **overall complexity** is not clearly defined
 - in **theory**, it could extend **indefinitely** (but the pseudo-random number generator or the memory configurations would yield an infinite loop)
 - in **practice**, it is defined by a **condition** imposed by the **user**

3 Effectiveness of a heuristic algorithm

As stated before, a heuristic algorithm is **useful** if it is:

- **efficient**: it “costs” much less than an exact algorithm
- **effective**: it “frequently” returns a solution “close to” an exact one

The **effectiveness** can be described in terms of:

- **closeness** of the solution to the **optimal one**
- **frequency** of **hitting optimal** or nearly optimal **solutions**

These features can be combined into a **frequency distribution** of **solutions** more or less close to the **optimum**.

The effectiveness can be **investigated** with a:

- **theoretical analysis** (*a priori*), **proving** that the **algorithm finds** always or with a given frequency **solutions** with a given guarantee of quality; this is done by looking at the structure of the algorithm
- **experimental analysis** (*a posteriori*), **measuring the performance** of the algorithm on sampled **benchmark instances** to show that it occurs

3.1 Distance

We need to define a distance to be able to **measure** the **effectiveness** of algorithms.

The effectiveness of a heuristic optimization algorithm A is measured by the **difference** between the heuristic **value** $f_A(i)$ and the **optimum** $f^*(i)$

- **Absolute difference**

$$\tilde{\delta}_A(i) = |f_A(i) - f^*(i)| \geq 0$$

used rarely and only when the objective is a pure number. This yields a small error on small instances and big numbers on big instances so it could be not significant.

- **Relative difference**

$$\delta_A(i) = \frac{|f_A(i) - f^*(i)|}{f^*(i)} \geq 0$$

frequent in experimental analysis (usually as a percent ratio).

- **Approximation ratio**

$$\rho_A(i) = \max \left[\frac{f_A(i)}{f^*(i)}, \frac{f^*(i)}{f_A(i)} \right] \geq 1$$

frequent in theoretical analysis: the first form is used for minimization problems, the second one for maximization ones. The formula is done this way to represent both problems with a single expression.

3.2 Theoretical analysis

3.2.1 Worst case

To obtain a compact measure, independent of i , find the worst case.

The **difference** between $f_A(i)$ and $f^*(i)$ is in general unlimited, but for some algorithms it is **limited**:

- **absolute approximation:**

$$\exists \tilde{\alpha}_A \in \mathbb{N} : \tilde{\delta}_A(i) \leq \tilde{\alpha}_A \text{ for each } i \in I$$

A (rare) example is Vizing's algorithm for Edge Coloring ($\tilde{\alpha}_A = 1$). The result will never be worse than the optimum by more than a fixed amount.

- **relative approximation:**

$$\exists \alpha_A \in \mathbb{R}^+ : \rho_A(i) \leq \alpha_A \text{ for each } i \in I$$

the ratio of the heuristic solution and the optimal one is not larger than a constant α (must be ≥ 1).

Factor α_A ($\tilde{\alpha}_A$) is the relative (absolute) **approximation guarantee**.

For other algorithms, the guarantee **depends** on the **instance size**

$$\rho_A(i) \leq \alpha_A(n) \text{ for each } i \in I_n, n \in \mathbb{N}$$

The approximation guarantee can change in relation to the instance size, (twice the size, twice as “less optimum” for example) so the guarantee can be expressed via a function.

Effectiveness can be **independent from size** (contrary to efficiency).

Achieving an approximation guarantee: For a minimization problem, the aim is to **prove** that

$$\exists \alpha_A \in \mathbb{R} : f_A(i) \leq \alpha_A f^*(i) \text{ for each } i \in I$$

We need to prove that A gives a **heuristic value** f_A on instance i that is **not larger** than a certain constant α times the optimal.

To **prove** it

1. find a way to build an **underestimate** $LB(i)$

$$LB(i) \leq f^*(i) \quad i \in I$$

for any instance, the underestimate must be smaller or equal to the optimum

2. find a way to build an **overestimate** $UB(i)$, related to $LB(i)$ by a **coefficient** α_A

$$UB(i) = \alpha_A LB(i) \quad i \in I$$

we found a lower bound, now we need an estimate for the upper bound, which will be at most α times the lower bound

3. find an **algorithm** A whose solution is not worse than $UB(i)$

$$f_A(i) \leq UB(i) \quad i \in I$$

Then $f_A(i) \leq UB(i) = \alpha A LB(i) \leq \alpha f^*(i)$, for each $i \in I$, we **proved** that

$$f_A(i) \leq \alpha_A f^*(i) \text{ for each } i \in I$$

our algorithm is at least α **times** the optimum.

A 2-approximated algorithm for the VCP: Given an **undirected graph** $G = (V, E)$ find the **minimum cardinality vertex subset** such that each **edge** of graph is **incident** to it.

A **matching** is a set of nonadjacent edges (they're connected to different vertices, they don't have common vertices).

Maximal matching is a matching such that any other edge of the graph is adjacent to one of its edges (it cannot be enlarged, any other edge is adjacent to one of the matching).

Matching algorithm:

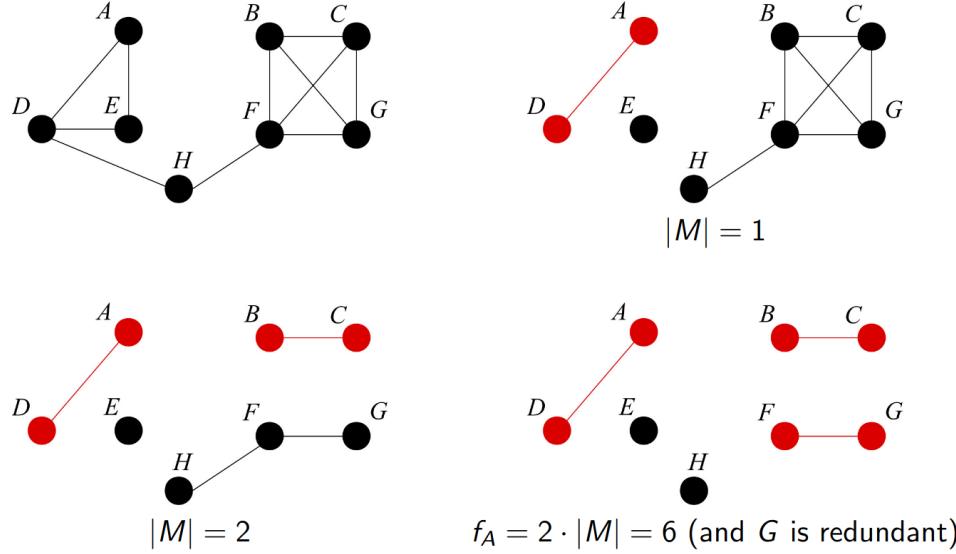
1. Build a **maximal matching** $M \subseteq E$ scanning the edges of E and including in M those not adjacent to M (now every edge of $E \setminus M$ is adjacent to an edge of M)
2. The **set of extreme vertices** of the matching edges is a VCP **solution**

$$x_A := \bigcup_{(u,v) \in M} \{u, v\}$$

and it can be improved removing the redundant vertices, as this can leave some unnecessary ones.

It basically takes all the nonadjacent edges and considers the relative vertices.

Example:



Proof: The matching algorithm is 2-approximated

1. The **cardinality** of matching M is an **underestimate** $LB(i)$
 - the cardinality of an **optimal covering** for any subset of edges $E' \subseteq E$ does not exceed that of an optimal covering for E
$$|x_{E'}^*| \leq |x_E^*|$$

(it costs more to cover all edges than only the matching, obviously since the matching is smaller)

 - the **optimal covering** of a **matching** M has **cardinality** $|M|$ (each edge of the matching requires exactly one different vertex, we're essentially considering half the vertices in the matching, which can cover all the relative edges)
2. **Including both the extremes** of each edge of the matching yields
 - an **overestimate** (it covers both the matching and all adjacent edges, i.e. a VCP solution for every maximal matching)
 - of **value** $UB(i) = 2LB(i)$ (two different vertices for each edge, double the cardinality which is our possible $LB(i)$)
3. The **matching algorithm** returns **solutions** of value $f_A(i) \leq UB(i)$ (possibly removing redundant vertices)

This **implies** $f_A(i) \leq 2f^*(i)$ for each $i \in I$, since $\alpha_A = 2$.

And the bound is tight: Since α_A relates $UB(i)$ and $LB(i)$, $f_A(i)$ and $f^*(i)$ could be closer.

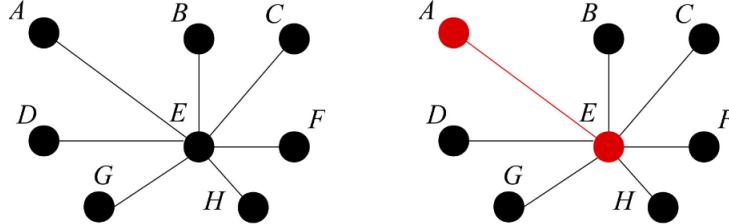
Actually, for many instances $\rho_A(i)$ is much better than α_A .

Are there instances \bar{i} for which $f_A(\bar{i}) = \alpha_A f^*(\bar{i})$? How are they like?

Our value $f_A(i)$ is **at most** α_A times the **optimum**, are there instances where our **solution** is exactly α_A times the optimal?

The study of these instances is useful to

- **evaluate** whether they are **rare or frequent**
- introduce **ad hoc modifications** to improve the algorithm



In this case our heuristic solution is exactly α_A (2) times the optimum, this is the worst case.

In the literature the typical expression “and the bound is tight” introduces the description of instances exhibiting the worst case.

If all worst cases are patched, the approximation guarantee improves.

The TSP under the triangle inequality: Consider the **TSP** with the additional (rather common) **assumptions** that

- graph $G = (N, A)$ is **complete**
- cost c is **nonnegative, symmetric** and **satisfies the triangle inequality**

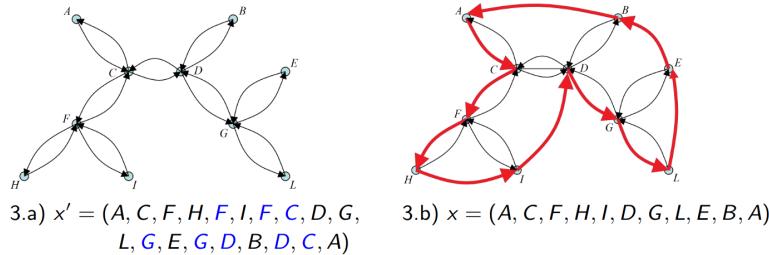
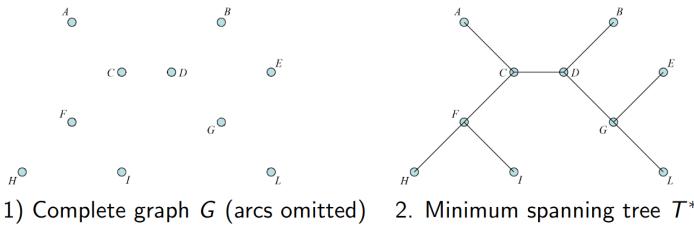
$$c_{ij} = c_{ji} \quad \forall i, j \in N \quad \text{and} \quad c_{ij} + c_{jk} \leq c_{ik} \quad \forall i, j, k \in N$$

i.e. the road works equally both ways and there can't be roads that connect i and j and subsequently k shorter than the road ik (the costs can form a triangle)

Double-tree algorithm

1. Consider the **complete undirected graph** corresponding to G
2. Build a **minimum cost spanning tree** $T^* = (N, X^*)$
3. Make a **pre-order visit** of T^* and build two lists of arcs:
 - a. x' lists the arcs used **both** by the **visit** and the **backtracking**: this is a circuit visiting each node, possibly several times; exactly twice the cost of the original minimum spanning tree, it's visiting all the nodes
 - b. x lists the arcs linking the **nodes in pre-order ending with the first**: this is a circuit visiting each node exactly once; we use the triangle inequality to go directly across nodes, removing some backtracking and visiting each node only once

Example:



The cost of the hamiltonian circuit will be at most equal to twice the spanning tree, which is at most equal to the optimal hamiltonian circuit. The **cost** is at most **twice the optimum**.

To do the last step you can just look at the path and skip already seen nodes, since it's a complete graph there will be a path to the next unseen node and, as stated by the triangle inequality, it will certainly be \leq than before.

This way the result is exactly an hamiltonian circuit, with a cost smaller than the double tree, which has a cost equal to twice the tree, which itself has a cost that smaller than the optimum.

The double-tree algorithm is **2-approximated**:

1. the **cost** of the **minimum spanning tree** is an **underestimate** $LB(i)$
 - **deleting** an **arc** from a Hamiltonian circuit yields a Hamiltonian **path** that is **cheaper**
 - a **Hamiltonian path** is a **spanning tree** (usually not of minimum cost)
2. the **cost** of circuit x' is
 - an **overestimate** $UB(i)$ (it is a non-minimum Hamiltonian circuit)
 - **equal** to $2LB(i)$ (two arcs correspond to each edge)
3. the **cost** of circuit x is $f_A(i) \leq UB(i)$ (a single direct arc replacing a sequence decreases the cost)

This **implies** that $f_A(i) \leq 2f^*(i)$ for each $i \in I$, so we get $\alpha_A = 2$.

Notice: x' is used in the approximation proof, but doesn't need to be computed.

Inapproximability: For an inapproximable problem, **approximated algorithms would be exact**.

Consider this family of TSP instances on **complete graphs**:

- $c_{ij} = 0$ for $(i, j) \in A_0$
- $c_{ij} = 1$ for $(i, j) \in (N \times N) \setminus A_0$ (the **triangle inequality is violated**, the earlier approximation can't be applied anymore)

Essentially, some arcs are 0, some 1, triangle inequality cannot hold.

The **optimum** of any such instance \bar{i} is:

$$\begin{cases} f^*(\bar{i}) = 0 & \text{if } A_0 \text{ contains a hamiltonian circuit} \\ f^*(\bar{i}) \geq 1 & \text{otherwise} \end{cases}$$

(in the latter case, the optimal solution contains at least an arc $\notin A_0$, i.e. an arc that costs 1).

Assume that a **polynomial algorithm** A provides a **guarantee** α_A

$$f^*(i) \leq f_A(i) \leq \alpha f^*(i) \quad \forall i \in I$$

Then $f^*(\bar{i}) = 0 \Leftrightarrow f_A(\bar{i}) = 0$.

Whenever the subgraph $G(N, A_0)$ has a Hamiltonian circuit, our algorithm A necessarily finds it, solving an NP-complete problem in polynomial time ($P = NP$, unlikely as this is of the millennium problems (I think)).

Approximation schemes: For hard problems

- **exact** algorithms provide the **best approximation guarantee** ($\alpha_A = 1$, they are exact after all), but require **exponential time** T_A
- **approximated** algorithms provide a **worse guarantee** ($\alpha_A > 1$), but could require **polynomial time** T_A

Some problems admit a **family of algorithms** providing a whole **range of compromises** between **efficiency** ed **effectiveness**

- better and better approximation guarantees: $\alpha_{A_1} > \dots > \alpha_{A_r}$
- worse and worse computational complexities: $T_{A_1} < \dots < T_{A_r}$

We can have multiple algorithms with different guarantees.

Approximation scheme is a parametric algorithm A_α allowing to choose α (Example: the KP).

3.2.2 Beyond the worst case

As usual, the worst-case approach is rough: some algorithms often have a good performance, though sometimes bad.

The **alternative approaches** are similar to the ones used for complexity

- **parametrisation:** prove an **approximation guarantee** that depends on **other parameters** of the instances besides the size n
- **average-case:** assume a **probability distribution** on the **instances** and evaluate the **expected value** of the approximation factor; the probability affects only the approximation factor, which is reasonable to assume can vary among instances (the algorithm could have a bad performance only on rare instances)

but there is at least **another approach**:

- **randomization:** the **operations** of the algorithm **depend** not only on the instance, but **also on pseudo-random numbers**, so that the **solution** becomes a **random variable** which can be investigated (the time complexity could also be random, but usually is not).

Random numbers are used to make decisions inside the algorithm, running it several time provides different results, the distribution of such results can be studied, usually in relation to the random seed used for each specific iteration.

3.2.3 Randomized approximation algorithms

For a randomised algorithm A , $f_A(i, \omega)$ and $\rho_A(i, \omega)$ are random variables depending on the pseudo-random number seed ω .

A randomised approximation algorithm has an approximation ratio whose expected value is limited by a constant

$$E[\rho_A(i, \omega)] \leq \alpha_A \text{ for each } i \in I$$

Randomized approximation for the MAX-SAT: given a CNF, find a truth assignment to the logical variables that satisfy a maximum weight subset of formulae.

Purely **random algorithm**: Assign to each variable xj ($j = 1, \dots, n$)

- value *False* with probability $1/2$
- value *True* with probability $1/2$

What is the **expected value** of the solution?

Let $\delta_i(x)$ be 1 if solution x satisfies clause i , 0 otherwise.

The **objective** $f(x) = f_A(I, \omega)$ is the **total weight** of the **satisfied clauses** and its expected value is

$$E[f_A(i, \omega)] = E\left[\sum_{i \in C} \delta_i(x) w_i\right] = \sum_{i \in C} (w_i \cdot \Pr[\delta_i(x) = 1])$$

The total sum is equal to the weight of the satisfied clauses (C contains all clauses, when not satisfied they are multiplied by $\delta_i(x) \implies 0$), this depends on the random seed, so the expected value is the weight of each clause multiplied by the probability of having a satisfied clause.

Our **probability** is a number between zero and one and we need to **estimate it**.

Let k_i be the number of literals of formula $i \in C$ and $k_{min} = \min_{i \in C} k_i$ (minimum number of literals among all the clauses)

$$Pr[\delta_i(x) = 1] = 1 - \left(\frac{1}{2}\right)^{k_i} \geq 1 - \left(\frac{1}{2}\right)^{k_{min}} \text{ for each } i \in C$$

$$\implies E[f_A(i, \omega)] \geq \sum_{i \in C} w_i \cdot \left[1 - \left(\frac{1}{2}\right)^{k_{min}}\right] = \left[1 - \left(\frac{1}{2}\right)^{k_{min}}\right] \sum_{i \in C} w_i$$

The first line essentially represents the probability of each clause not having a single “good” literal, which will at least be $1 - (1/2)^{k_{min}}$ since k_{min} is the minimum number of literals \implies highest probability of not satisfying the clause.

The second row is a minorization of the probability of each clause (kinda like an underestimate of the probability).

And since $\sum_{i \in C} w_i \geq f^*(i)$ for each $i \in I$ **one obtains**

$$\frac{E[f_A(i, \omega)]}{f^*(i)} \geq \left[1 - \left(\frac{1}{2}\right)^{k_{min}}\right] \geq \frac{1}{2}$$

This is an incredibly simple algorithm, but the sample average of many iterations will tend to get near the theoretical the expected value, but the best of the many iterations will certainly better than the average and consequently better than the theoretical approximation.

3.3 Empirical analysis

The theoretical analysis is **complicated** by the fact that

- the steps of the algorithm have a **complex effect** on the **solution** though usually **not** on the **computational cost**
- **average case** and **randomization** require a **statistical treatment**

The theoretical analysis can be **unsatisfactory** in practice when its conclusions are based on **unrepresentative assumptions**

- an **infrequent worst case** (very hard and very rare instances)
- an **unrealistic** probability **distribution** of the instances

Experimental analysis chooses a **benchmark** of instances and **measures performance** on that specific set (obviously not all possible instances).

The experimental approach is very common in science

- mathematics is an exception, based on the formal approach
- algorithmics is an exception within the exception, you can get useful information from practical performances

The basics of the **experimental approach** are

1. start from **observation**
 2. formulate a **model** (work hypothesis)
 3. repeat the following steps
 - a. **design** computational **experiments** to validate the model
 - b. **perform** the experiments and collect their results
 - c. **analyze** the **results** with quantitative methods
 - d. **revise** the **model** based on the results
- until a **satisfactory model** is obtained

What is a “**model**” in the study of algorithms?

- in physics the laws that rule the behavior of phenomena, an assumption about the physical law that affects a certain phenomenon
- in **algorithmics** the **laws** that **rule the behavior** of algorithms, assumptions on the laws that an algorithm follows

The **experimental analysis** of algorithms **aims** to

- obtain **compact indices of efficiency** and **effectiveness** of an algorithm
- **compare the indices** of different algorithms to rank them
- **describe the relation** between the performance **indices and parametric values** of the instances (size n , etc...)
- **suggest improvements** to the algorithms

I can assume that my algorithm will be linear/quadratic/etc in respect of the size/any other parameter, these are assumptions based on the knowledge of the algorithm and even experiments. This can give a descriptive model regarding the behavior of the algorithm, leading to some technological modifications to improve the algorithm itself.

3.3.1 Benchmark sample

As not all instances can be tested, a benchmark sample must be defined.

A meaningful **sample** must **represent different**

- **sizes**, in particular for the analysis of the computational cost; size largely determines the complexity and possibly the quality of the solution
- **structural features** (in the case of graphs: density, degree, diameter, ...; any type of index regarding a specific structure)
- **types**
 - of **application**: logistics, telecommunications, production, ...
 - of **generation**: realistic, artificial, transformations of other problems
 - of **probabilistic distribution**: uniform, normal, exponential, ...
instances can come from different sources, affecting efficiency

Looking for an “**equiprobable**” benchmark sample **is meaningless** because the instance sets are infinite and infinite sets do not admit equiprobability (big statistic question).

On the contrary, we can **define** finite **classes of instances** that are

- **sufficiently hard** to be instructive
- **sufficiently frequent** in applications to be of interest
- **quick enough** to solve to provide sufficient data for inferences

and **extract benchmark samples** from these classes.

Reproducibility: The scientific method requires **reproducible and controllable results**

- concerning the **instances**, one must use
 - publicly available instances
 - new instances made available to the community
- concerning the **algorithm**, one must specify
 - all implementation details
 - the programming language
 - the compiler
- concerning the **environment**, one must specify
 - the machine used
 - the operating system
 - the available memory
 - ...

Every detail about the instance must be specified to get the same results, even seemingly negligible details can matter in some instances (e.g. different compilers should provide the same result but may yield somewhat different performances).

Reproducing results obtained by others is always extremely difficult, these are guidelines but there are limitations (reproducing the machine for example).

3.3.2 Comparing heuristic algorithms

A heuristic algorithm is **better** than another one when it **simultaneously**

- obtains **better results**
- requires a **smaller time**

Slow algorithms with good results and fast algorithms with bad results **cannot be compared in a meaningful way** (what if the second had yielded better results if given more time?).

It can be justified to **neglect the computational time** when

- considering a **single algorithm** with no **comparison**
- comparing algorithms that perform the **same operations** (e.g., variants of the same algorithm obtained modifying a numerical parameter)
- comparing algorithms that mostly perform the same operations with **few different ones** that take a **negligible fraction of the time** (e.g., different initializations or perturbations)

A statistical model of algorithm performance: the idea is modeling the execution of algorithm A as if it was a random experiment

- the whole set of **instances** I is the sample space
- the benchmark **subset** of instances $\bar{I} \subset I$ is the **sample**
- the **computational time** $T_A(i)$ is a random variable; the time required by algorithm A on instance i
- the **relative difference** $\delta_A(i)$ is a random variable; the difference of the result obtained by A on i with the optimal result, divided by the optimal result

and the statistical properties of the random variables $T_A(i)$ and $\delta_A(i)$ describe the **performance** of A . Both $T_A(i)$ and $\delta_A(i)$ are random variables because they depend on the instance.

Instead of considering all the possible values of the computational time and relative difference, we are describing the random variables $T_A(i)$ and $\delta_A(i)$, with their statistical properties, their description will give a description of the performances of the algorithm.

Estimates of $\delta_A(i)$: The computation of $\delta_A(i)$ requires to know the optimum $f^*(i)$:

$$\delta_A(i) = \frac{|f_A(i) - f^*(i)|}{f^*(i)}$$

What if the **optimum** is **unknown**? If we can't have an exact optimum value, we need to get an estimate (form below or above, depending on if it's a minimization or maximization problem).

Replace it with an underestimate $LB(i)$ and/or an overestimate $UB(i)$

$$LB(i) \leq f^*(i) \leq UB(i) \implies \frac{1}{LB(i)} \geq \frac{1}{f^*(i)} \geq \frac{1}{UB(i)} \implies$$

$$\implies \frac{f_A(i)}{LB(i)} - 1 \geq \frac{f_A(i)}{f^*(i)} - 1 \geq \frac{f_A(i)}{UB(i)} - 1$$

$$\frac{f_A(i)}{f^*(i)} - 1 = \begin{cases} \delta_A(i) & (\text{minimization}) \\ -\delta_A(i) & (\text{maximization}) \end{cases} \implies \frac{\frac{f_A(i) - UB(i)}{UB(i)}}{\frac{f_A(i) - LB(i)}{LB(i)}} \leq \delta_A(i) \leq \frac{\frac{f_A(i) - LB(i)}{LB(i)}}{\frac{f_A(i) - UB(i)}{UB(i)}}$$

and therefore

$$\frac{|f_A(i) - UB(i)|}{UB(i)} \leq \delta_A(i) \leq \frac{|f_A(i) - LB(i)|}{LB(i)}$$

This range yields a region estimate for the SQD diagram.

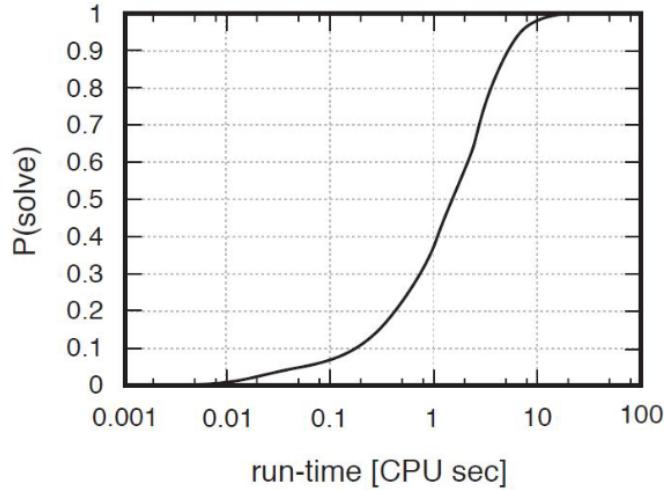
3.3.3 Analysis of the computational time

The Run Time Distribution (RTD) diagram: the RTD is the plot of the **distribution function** of $T_A(i)$ on \bar{I} (time required on an instance i on the given benchmark)

$$F_{T_A}(t) = \Pr [T_A(i) \leq t] \text{ for each } t \in \mathbb{R}$$

It gives the **probability** that the **execution time** is below a time t .

Since $T_A(i)$ strongly depends on the size $n(i)$, meaningful RTD diagrams usually refer to **benchmarks** \bar{I}_n with **fixed** n (and possibly other fixed parameters suggested by the worst-case analysis).

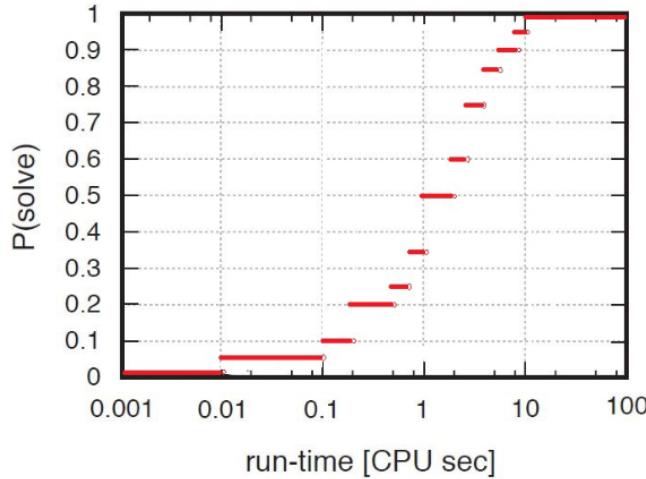


If given only n as a parameter there are really slow and really fast instances, there could be another parameter which plays a role in the execution time. Essentially dividing \bar{I}_n in further “sub-classes”.

If all influential parameters are identified and fixed, the RTD diagram degenerates into a step function (all instances require the same time; zero probability under a value, 1 over).

The Run Time Distribution (RTD) diagram is

- **monotone nondecreasing:** more instances are solved in longer times
- **step-wise and right-continuous:** the graph steps up at each $T(i)$
- **equal to zero for $t < 0$:** no instance is solved in negative time
- **equal to 1 for $t \geq \max_{i \in \bar{I}} T(i)$:** all are solved within the longest time



For large benchmark samples, the plot looks continuous, but it is not.

The graph goes up by steps because it shows how many instances (in percent) are solved given a certain amount of time.

If there is a big enough number of instances the number of “steps” will increase, giving the impression of a continuous graph.

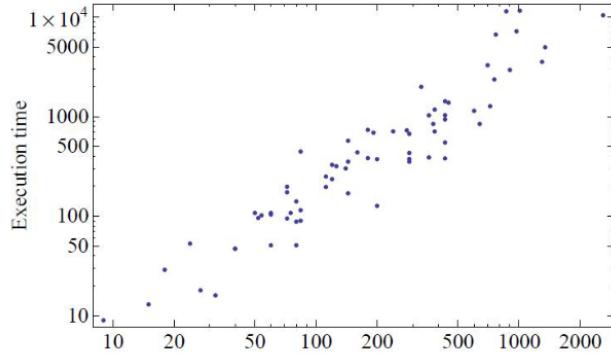
In order to **build the diagram**:

1. **run the algorithm on each instance $i \in \bar{I}$**
2. **build the set $T_A(\bar{I}) = \{T_A(i) : i \in \bar{I}\}$** (it's just a set of numbers)
3. **sort $T_A(\bar{I})$ by non-decreasing values: $t_1 \leq \dots \leq t_{|\bar{I}|}$**
4. **plot the points $(t_j, \frac{j}{|\bar{I}|})$** for $j = 1, \dots, |\bar{I}|$ (for each t_j , only the highest) and the horizontal segments (close on the left, open on the right)

Scaling diagram: this diagram describes the dependence of $T(i)$ on the size $n(i)$

- **generate a sequence of values** of n and a sample \bar{I}_n for each value (generate a benchmark sample for each size)
- **apply the algorithm** to each $I \in \bar{I}_n$ for all n (all instances of each sub-benchmark for all the sizes)
- **sketch** all points $(n(i), T(i))$ or the mean points $\left(n, \frac{\sum_{i \in \bar{I}_n} T(i)}{|\bar{I}_n|}\right)$
- assume an **interpolating function**
- estimate the **numerical parameters** of the interpolating function

It shows how the computational time scales in relation to the size of the instance.



This analysis provides an **empirical average-case complexity**

- with **well-determined multiplying factors** (instead of c_1 and c_2)
- **not larger than the worst-case one** (it also includes easy instances)

The correct family of interpolating functions can be suggested

- by a **theoretical analysis** (studying the algorithm)
- by **graphical manipulations**

(Linear interpolation is usually the right tool).

The **scaling diagram turns into a straight line** when

- an **exponential algorithm** is represented on a **semi-logarithmic scale** (the logarithm is applied only to the time axis)

$$\log_2 T(n) = \alpha n + \beta \Leftrightarrow T(n) = 2^\beta (2^\alpha)^n$$

if the log of the time is a linear function of the size, necessarily the algorithm is exponential.

I see the time goes up a lot, I have reason to suspect the algorithm is exponential from the theoretical analysis \Rightarrow I try to use a logarithmic scale for the time.

- a **polynomial algorithm** is represented on a **logarithmic scale** (the logarithm is applied to both axes)

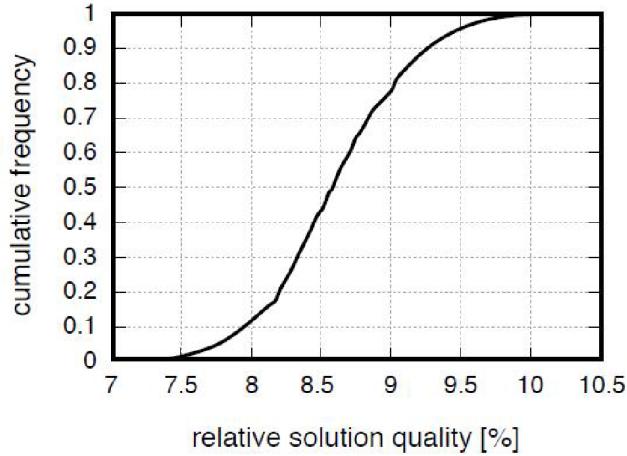
$$\log_2 T(n) = \alpha \log_2 n + \beta \Leftrightarrow T(n) = 2^\beta n^\alpha$$

the log of the time is a log function of the size, then the computational time is polynomial (α is the exponent).

3.3.4 Analysis of the quality of the solution

The **Solution Quality Distribution (SQD) diagram** is the plot of the **distribution function** of $\delta_A(i)$ on \bar{I} (relative difference on the benchmark)

$$F_{\delta_A}(\alpha) = \Pr [\delta_A(i) \leq \alpha] \text{ for each } \alpha \in \mathbb{R}$$

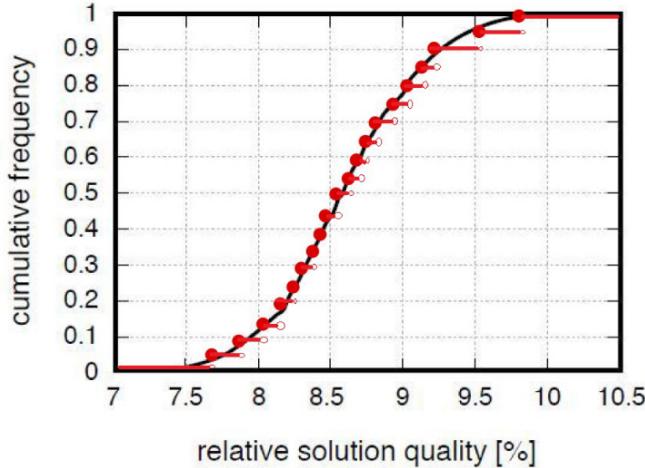


It plots the probability that the relative difference is smaller than a value α , for each possible value of α .

One axis is the frequency, the other is the relative difference, it basically answers the question “how many instances got this δ_A or less?”.

For any algorithm, the **distribution function** of $\delta_A(i)$ is

- **monotone non-decreasing:** more instances are solved with worse gaps
- **step-wise and right-continuous:** the graph steps up at each $\delta(i)$
- **equal to zero for $\alpha < 0$:** no instance is solved with negative gap
- **equal to 1 for $\alpha \geq \max_{i \in \bar{I}} \delta(i)$:** all are solved within the largest gap



As with the RTD, the more instances you have the more this diagram looks continuous.

If A is an

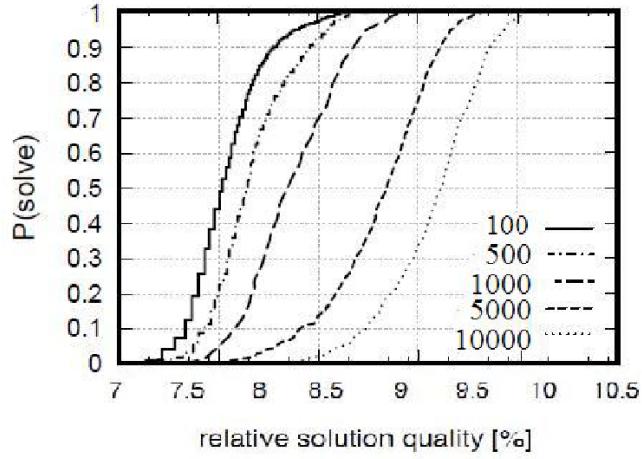
- exact algorithm, it is a step-wise function, equal to 1 for all $\alpha \geq 0$
- $\bar{\alpha}$ -approximated algorithm, it is a function equal to 1 for large α

In order to **build the diagram**:

1. **run the algorithm on each instance** $i \in \bar{I}$
2. **build the set** $\Delta_A(\bar{I}) = \{\delta_A(i) : i \in \bar{I}\}$, the relative difference for each instance (you need the optimum, could be approximated)
3. **sort** $\Delta_A(\bar{I})$ non-decreasing values: $\delta_1 \leq \dots \leq \delta_{|\bar{I}|}$
4. **plot points** $(\delta_j, \frac{j}{|\bar{I}|})$ for $j = 1, \dots, |\bar{I}|$ (for each δ_j , only the highest) and the horizontal segments (close on the left, open on the right)

Parametric SQD diagrams: Given the theoretical and practical problems to build a **meaningful sample** often the **diagram is parameterized** with respect to

- a **descriptive parameter of the instances** (size, density, ...)
- a **parameter of the probability distribution assumed** for the **instances** (expected value or variance of the costs, ...)



The conclusions are more limited, but the **sample is more significant**. General trends can be highlighted (what happens as size increases?).

It's basically dividing the benchmark set into sub-benchmarks, following some parameters (such as size, ecc...), to get the solution quality in relation to those parameters.

Comparison between algorithms with the SQDs: How to determine whether an **algorithm is better than another?** (for simplicity's sake, the algorithms take the same time)

- **strict dominance:** it obtains better results on all instances

$$\delta_{A_2}(i) \leq \delta_{A_1}(i) \text{ for each } i \in I$$

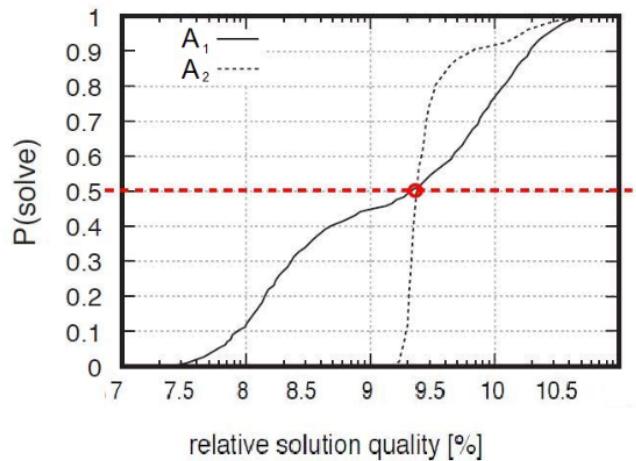
This usually happens only in trivial cases (e.g., A_2 “includes” A_1). A_2 is always better.

- **probabilistic dominance:** the distribution function has higher values for every value of α

$$F_{\delta_{A_2}}(\alpha) \geq F_{\delta_{A_1}}(\alpha) \text{ for all } \alpha \in \mathbb{R}$$

the SQD of A_2 is larger or equal than the one for A_1 (is “above”, if you set a certain threshold of “quality” A_2 has a better occurrence rate).

The following plot shows no dominance, but A_1 is less “robust” than A_2 : A_1 has results more dispersed than A_2 (both better and worse)



3.3.5 Compact statistical descriptions

The distribution function F_{δ_A} can be replaced or accompanied by more compact characterizations of the effectiveness of an algorithm.

This typically involves classical **statistical indices** of

- **position**, such as the sample mean

$$\bar{\delta}_A = \frac{\sum_{i \in \bar{I}} \delta_A(i)}{|\bar{I}|}$$

- **dispersion**, such as the sample variance

$$\bar{\sigma}_A^2 = \frac{\sum_{i \in \bar{I}} (\delta_A(i) - \bar{\delta}_A)^2}{|\bar{I}|}$$

These indices “**suffer**” from the influence of **outliers**.

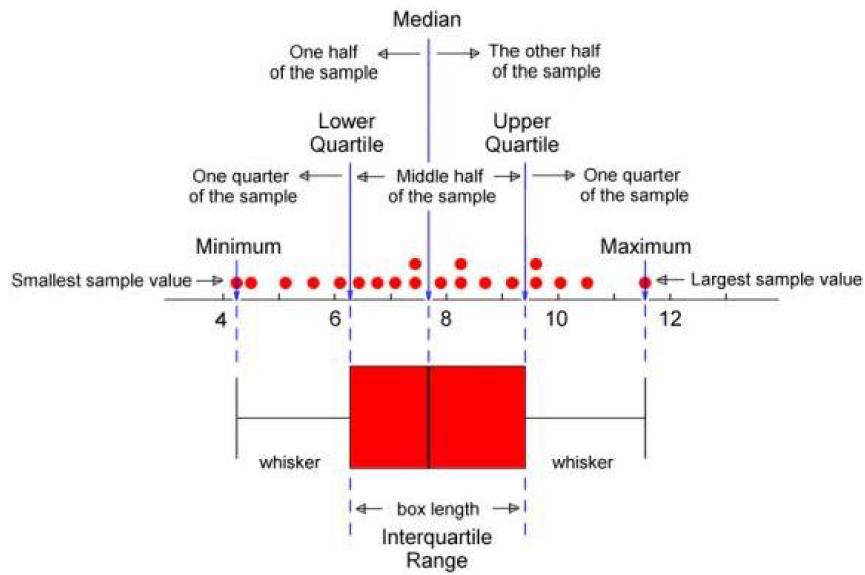
Other statistical indices are “stabler” and more detailed

- the **sample median**
- suitable **sample quantiles**

3.3.6 Boxplots diagrams

Boxplot: A graphic representation is the boxplot (or box and whiskers diagram)

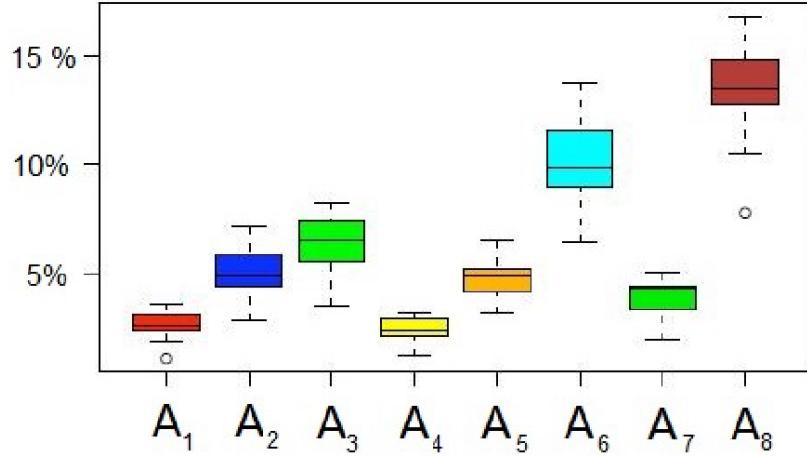
- sample **median** ($q_{0.5}$)
- **lower** and **upper** sample **quartiles** ($q_{0.25}$ and $q_{0.75}$)
- the **extreme** sample values (often excluding the “outliers”)



On the x axis there's the relative difference (in the example, from 4 to 12). The median is in the middle, the box represents the two middle quartiles, the whiskers represent the last two quartiles.

To have a boxplot diagram you need to have: extreme values (upper and lower), lower and upper quartiles, median. If you have these values you can draw the diagram.

Comparison between algorithms with boxplot diagrams: A more compact comparison can be performed with boxplot diagrams (the dots represent outliers)



Necessary conditions:

$$\begin{aligned} \text{Strict dominance} &\Rightarrow \text{Probabilistic dominance} \\ &\Rightarrow q_i \leq q'_i (i = 1, \dots, 5) \end{aligned}$$

Strict dominance holds only if probabilistic dominance holds.

Strict dominance holds if a boxplot is **fully below the other** (e.g. $A_7 - A_8$).

Probabilistic dominance holds only if **each of the five quartiles is not above** the corresponding **one of the other** algorithm (e.g. $A_2 - A_3$).

Strict dominance means a box is fully below another, probabilistic dominance means a box is at least partially below another. Obviously the first implies the latter.

3.3.7 Relation between quality and computational time

Many heuristic algorithms find **several solutions** during their execution, instead of a single one, and consequently can be **terminated prematurely**.

In particular, **metaheuristics** (random steps or memory mechanisms) have a computational time t fixed by the user and potentially unlimited; they often find a solution and improve on it.

Let $\delta_A(i, t)$ be the **relative difference** reached by A at time t on instance i .

As a function of time t , $\delta_A(i, t)$ is

- $+\infty$ if A has **not yet found** a feasible **solution** at time t
- step-wise **monotone nonincreasing** (the solution doesn't get worse)
- **constant** after the regular **termination** ($t \geq T(i)$; after the termination of the algorithm the solution doesn't improve)

Randomized algorithms: For randomized algorithms the relative difference $\delta_A(i, \omega, t)$ depends on

1. the **instance** $i \in I$
2. the **outcome** $\omega \in \Omega$ of the random experiment guiding the algorithm (the random seed)
3. the **execution time** t

Given a **fixed time**, these algorithms can be **tested**

1. on a **sample of instances** \bar{I} with a **fixed seed** ω
2. on a **fixed instance** i with a **batch of seeds** $\bar{\Omega}$ (different runs)
3. on **several instances** with **several seeds** on each instance

The results of **multiple runs** ($\bar{\Omega}$) are usually **summarized** providing both:

- the **minimum relative difference** $\delta_A^*(i, t)$ and the **total time** $|\bar{\Omega}|t$
- the **average relative difference** $\bar{\delta}_A(i, t)$ and the **single-run time** t

3.4 Classification

The relation between solution quality and computational time allows to **classify the algorithms** into:

- **complete:** for each instance $i \in I$, find the optimum in finite time

$$\exists \bar{t}_i \in \mathbb{R}^+ : \delta_A(i, t) = 0 \text{ for each } t \geq \bar{t}, i \in I$$

(Another name for exact algorithms) for every instance the algorithm gives the optimum in finite time.

- **probabilistically approximately complete:** for each instance $i \in I$, find the optimum with probability converging to 1 as $t \rightarrow +\infty$

$$\lim_{t \rightarrow +\infty} Pr[\delta_A(i, t) = 0] = 1 \text{ for each } i \in I$$

(Many randomized metaheuristics) for all instances, the probability of getting an exact solution converges to 1, given enough time.

- **essentially incomplete:** for some instances $i \in I$, find the optimum with probability strictly < 1 as $t \rightarrow +\infty$

$$\exists i \in I : \lim_{t \rightarrow +\infty} Pr[\delta_A(i, t) = 0] < 1$$

(Most greedy algorithms, local search algorithms, ...) even given infinite time, some instances can't get to the exact solution.

3.4.1 Beyond the optimum, a generalization

An obvious generalization replaces the search for the **optimum** with that for a **given level of approximation** $A(I, t) = 0 \rightarrow A(I, t)$

$$\delta_A(i, t) = 0 \rightarrow \delta_A(i, t) \leq \alpha$$

We're not searching for an exact solution, we're looking for a "good enough" solution, where the "good enough" is determined by α .

The algorithms can now be **classified** in

- **α -complete algorithms:** for each instance $i \in I$, find an α -approximated solution in finite time (α -approximated algorithms).
- **probabilistically approximately α -complete algorithms:** for each instance $i \in I$, find an α -approximated solution with probability converging to 1 as $t \rightarrow +\infty$.
- **essentially α -incomplete algorithms:** for some instances $i \in I$, find an α -approximated solution with probability strictly < 1 as $t \rightarrow +\infty$

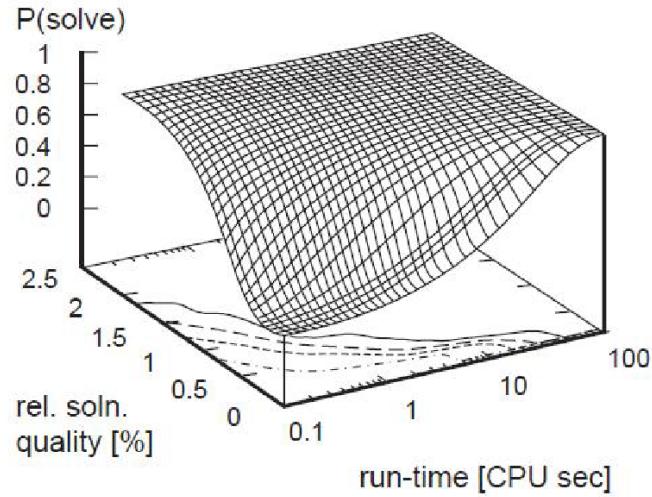
In conclusion, every algorithm provides a **compromise** between

- a **quality measure**, described by the threshold α
- a **time measure**, described by the threshold t

3.4.2 Probability of success

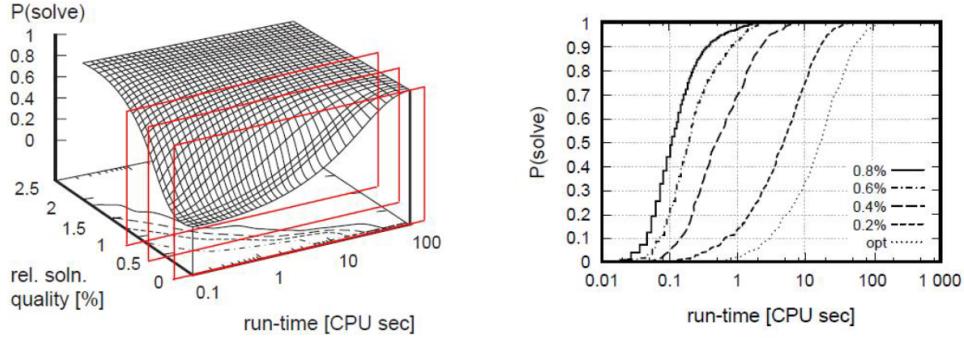
Let the **success probability** $\pi_{A,n}(\alpha, t)$ be the probability that algorithm A finds in **time** $\leq t$ a solution with a **gap** $\leq \alpha$ on an instance of **size** n

$$\pi_{A,n}(\alpha, t) = \Pr [\delta_A(i, t) \leq \alpha | i \in I_n, \omega \in \Omega]$$



This yields different secondary diagrams, derived from the time, solution quality and probability of solving.

Qualified Run Time Distribution (QRTD) diagrams: The QRTD diagrams describe the profile of the time required to reach a specified level of quality.



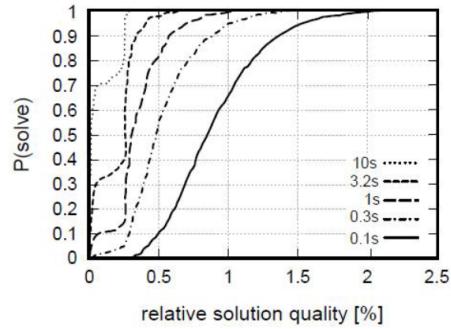
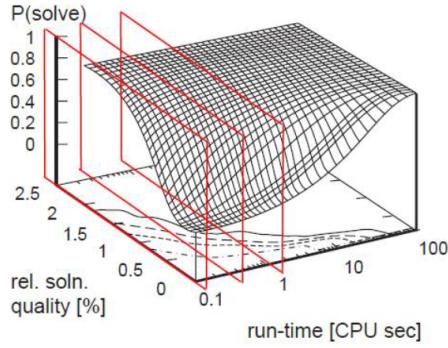
This is fixing the quality of the solution α and showing how the probability of getting such quality changes in relation to computational time.

They are useful when the computational time is not a tight resource.

If the algorithm is

- complete, all diagrams reach 1 in finite time
- $\bar{\alpha}$ -complete, all diagrams with $\alpha \geq \bar{\alpha}$ reach 1 in finite time
- $\bar{\alpha}$ -incomplete, all diagrams with $\alpha \leq \bar{\alpha}$ do not reach 1

Timed Solution Quality Distribution (TSQD) diagrams: The TSQD diagrams describe the profile of the **level of quality** reached in a **given computational time**



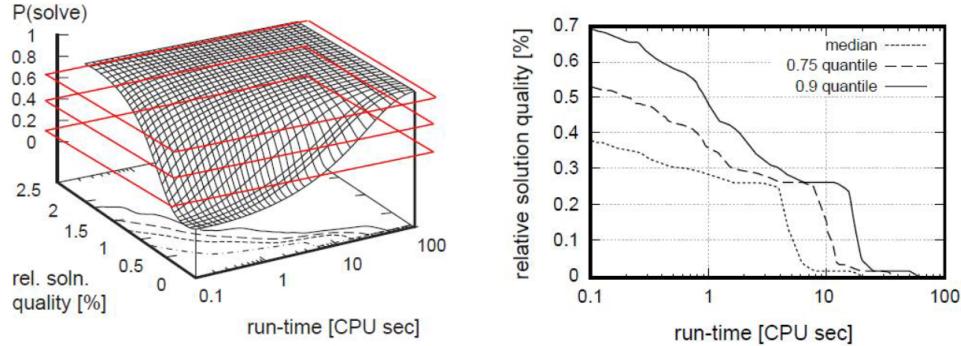
This is fixing the computational time and showing what's the probability of getting a certain quality of solution in that given time.

They are useful when the computational time is a tight resource.

If the algorithm is

- complete, all diagrams with a sufficient t are step functions in $\alpha = 0$
- $\bar{\alpha}$ -complete, all diagrams with a sufficient t reach 1 in $\alpha = \bar{\alpha}$
- probab. approx. $\bar{\alpha}$ -complete, the diagrams converge to 1 in $\alpha = \bar{\alpha}$
- $\bar{\alpha}$ -incomplete, all diagrams keep < 1 in $\alpha = \bar{\alpha}$

Solution Quality statistics over Time (SQT) diagrams: One can also draw the **level lines** associated to different quantiles.



This shows how the solution quality changes in relation to the computational time, given a fixed probability of getting such quality of solution.

They describe the compromise between quality and computational time.

For a robust algorithm the level lines are very close to each other.

3.5 Statistical tests (Wilcoxon's test)

Diagrams and boxplots are qualitative: how do you **evaluate** quantitatively if the **empirical difference between algorithms** A_1 and A_2 is **significant**?

Wilcoxon's test focuses on effectiveness (neglecting robustness)

- $f_{A_1}(i) - f_{A_2}(i)$ is a **random variable defined on the sample space** I , objective function, measures the quality; if it's negative A_1 is better than A_2 and vice versa (assuming a minimization problem).
 - Formulate a **null hypothesis** H_0 according to which the **theoretical median** of $f_{A_1}(i) - f_{A_2}(i)$ is **zero** (they are equivalent, half the time one is better, switching for the other half).
 - Extract a **sample of instances** \bar{I} and **run the two algorithms** on it, obtaining a **sample of pairs of values** (f_{A_1}, f_{A_2}) .
 - Compute the **probability** p of **obtaining the observed result** (that particular set of pairs) or a more “extreme” one (that set or less probable), assuming that H_0 is true (assuming it follows our hypothesis); is it just bad luck or is the hypothesis wrong?
 - Set a **significance level** \bar{p} , that is the
 - **maximum acceptable probability to reject** H_0 assuming that it is true
 - that is, to consider **two identical medians as different**
 - that is, to consider **two equivalent algorithms as differently effective** (referring to the median of the gap)
- \bar{p} is the probability above which we can say that the hypothesis follows our observed results.
- **reject** H_0 **when** $p < \bar{p}$

Typical values for the significance level are $\bar{p} = 5\%$ or $\bar{p} = 1\%$.

3.5.1 Assumptions

It is a **nonparametric** test, that is, it does **not** make **assumptions** on the **probability distribution** of the tested **values**.

It is useful to evaluate the performance of heuristic algorithms, because the **distribution of the result** $f_A(i)$ is **unknown**.

It is based on the following **assumptions**:

- all **data** are **measured** at least on an **ordinal scale** (the specific values do not matter, only their relative size, you just need a sequence of levels, we use numbers so it's obviously verified)
- the two **data sets** are **matched** and **derive from the same population** (we apply A_1 and A_2 to the same instances, extracted from I , so it's always verified)
- each **pair of values is extracted independently** from the others (the instances are generated independently from one another, not always satisfied)

3.5.2 Application

Application of the Wilcoxon's test:

1. Compute the **absolute differences** $|f_{A_1}(i_j) - f_{A_2}(i_j)|$ for all $i_j \in \bar{I}$ (instance by instance).
2. **Sort** them by **increasing** values and assign a **rank** R_j to each one; all zero-differences can be removed; in case of ties, all should have an average.
We're getting the sum of all the ranks, which is proportional to the difference, this serves the purpose of considering if the difference itself is large or small and not only the number of instances an algorithm has "won".
3. Separately **sum the ranks of the pairs** with a positive difference and those of the pairs with a negative difference

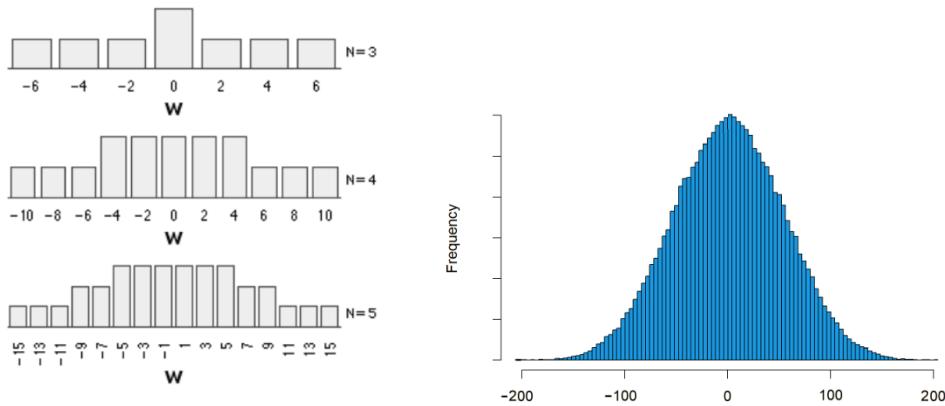
$$\begin{cases} W^+ &= \sum_{j: f_{A_1}(i_j) > f_{A_2}(i_j)} R_j \\ W^- &= \sum_{j: f_{A_1}(i_j) < f_{A_2}(i_j)} R_j \end{cases}$$

If the null hypothesis H_0 were true, the two sums should be equal. Each positive is summed together, same with the negatives, if the algorithms are truly equivalent (as per the null hypothesis), these should be the same, i.e. $W^+ - W^-$ should be 0. We're computing the sum of the ranks for the instances where the first algorithm is better than the second and vice versa.

4. The **difference** $W^+ - W^-$ allows to **compute the value of p** : each of the $|\bar{I}|$ differences can be positive or negative: $2^{|\bar{I}|}$ outcomes; p is the fraction with $|W^+ - W^-|$ equal or larger than the observed value.
5. if $p < \bar{p}$, the **difference is significant** and
 - if $W^+ < W^-$, A_1 is better than A_2
 - if $W^+ > W^-$, A_1 is worse than A_2

Computation of the p -value: The value of p (distribution of probability) is usually

- **computed explicitly** by enumeration when $|\bar{I}| < 20$; which possible cases give us each value?
- **approximated with a normal distribution** when $|\bar{I}| \geq 20$; there's a lot of cases, it can be approximated



Of course, pre-computed tables also exist.

We get the probability distribution and we observe the probability of the value we got, then we need to evaluate if it's reasonable or if it's a significant result.

Possible conclusions: Wilcoxon's test can suggest

- that **one of the two algorithms is significantly better** than the other
- that the **two algorithms are statistically equivalent** (but take it as a stochastic response, and keep an eye on p)

If the sample includes instances of different kinds, two algorithms could be **overall equivalent**, but **nonequivalent** on the **single classes** of instances.

Dividing the sample could reveal

- classes of instances for which A_1 is better
- classes of instances for which A_2 is better
- classes of instances for which the two algorithms are equivalent

but multiplying questions means getting some wrong answers by chance (FWER = Family-Wise Error Rate).

What about testing $\delta_A(i)$ instead of $f_A(i)$? (Open question).

4 Constructive Heuristics

In Combinatorial Optimization every solution x is a subset of the ground set B .

Essentially we want to add an element at a time to an empty subset until it doesn't make sense anymore to have new elements.

A constructive heuristic **updates a subset $x^{(t)}$ step by step**

1. **start** from an **empty subset**: $x^{(0)} = \emptyset$ (obviously a subset of any optimal solution)
2. **stop** when a **termination condition** holds (the following subsets cannot be optimal solutions, there are no more elements to be added that lead to an optimal solution)
3. **select** the “**best**” element $i^{(t)} \in B \setminus x$ among the “acceptable” ones (parameter which needs to be determined) at the current step t (try and keep $x^{(t)}$ within a feasible and optimal solution)
4. **add** $i^{(t)}$ to the **current subset** $x^{(t)}$: $x^{(t+1)} := x^{(t)} \cup \{i^{(t)}\}$ (the selection can never be undone!)
5. go **back to point 2**

Such processes admit a nice modeling tool.

4.1 Construction Graph

Every construction heuristic A defines a **construction graph**

- the **node set** $F_A \subseteq 2^B$ (**search space**) is the collection of all subsets $x \subseteq B$ acceptable for A
- the **arc set** is the collection of all pairs $(x, x \cup \{i\})$ such that $x \in F_A$, $i \in B \setminus x$ and $x \cup \{i\} \in F_A$ (connects x with x plus one element). The arcs represent the elementary extensions of the acceptable subsets

It's a directed graph in which the nodes compose the search space, dependent on A , and the arcs link all the elements of the search space to every element with a cardinality of one more. Essentially the nodes are all the possible subsets, while the arcs represent all the possible connections (additions of elements) for each node (subset).

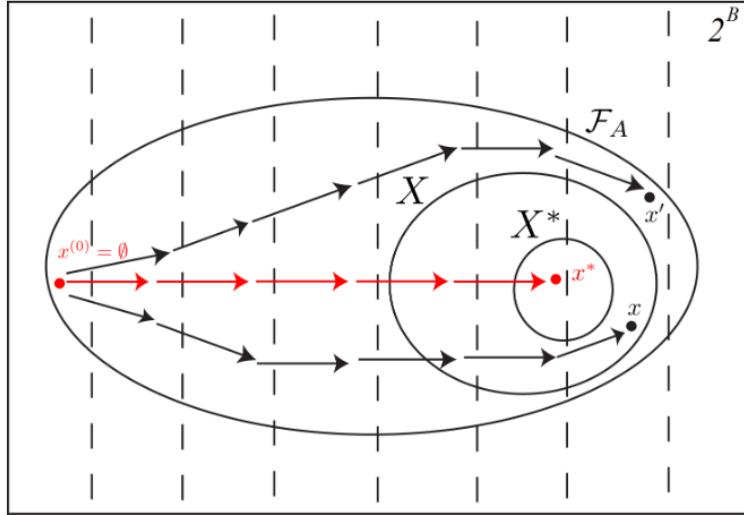
The construction graph is by definition **acyclic**.

Each possible execution of A is a **maximal path of the construction graph**

- **from** the empty subset $x = \emptyset$
- to a subset x that **cannot be acceptably extended** (and hopefully a feasible solution)

$$\Delta_A^+(x) = \{i \in B \setminus x : x \cup \{i\} \in F_A\} = \emptyset$$

Example



The algorithm visits a **sequence of subsets** $\emptyset = x^{(0)} \subset \dots \subset x^{(t_f)}$ terminating

- in an **optimal solution** $x^* \in X^*$ (Example: MST problem, both with $F_{Kruskal}$ and F_{Prim})
- in a **nonoptimal feasible solution** $x \in X$ (Example: KP, MDP, etc...)
- in an **unfeasible subset** x' (Example: TSP on a noncomplete graph)

4.1.1 Termination condition

A constructive heuristic A **terminates** when

- the current **subset** $x^{(t)}$ has **no outgoing arc**

$$\Delta_A^+(x^{(t)}) = \{i \in B \setminus x^{(t)} : x^{(t)} \cup \{i\} \in F_A\} = \emptyset$$

$\Delta_A^+(x^{(t)})$ represents the subset of the ground set that can be added in $x^{(t)}$ in an acceptable manner for algorithm A (what I can add to our solution).

- that is, **extending** $x^{(t)}$ implies to **leave the search space** F_A

$$x^{(t)} \cup \{i\} \notin F_A \text{ for each } i \in B \setminus x^{(t)}$$

There is no arc going out of the node in the construction graph.

The algorithm returns the **best solution visited** during the execution (usually the last one).

Different behaviors are possible

- sometimes **all** visited **subsets** are **feasible** (e.g., KP)
- often the **last subset** is the only **feasible** solution
- $x^{(t)}$ could even **move in and out** of X (or X^* ; but this is uncommon)
- the path **can visit or not** X and X^*

4.1.2 Definition of the construction graph

Ideally, the **search space** F_A should include

- the **empty subset**: $\emptyset \in F_A$ (A starts from \emptyset)
- all **feasible solutions**: $X \subseteq F_A$ (maybe excluding provably nonoptimal solutions)
- **only subsets accessible from \emptyset** (inaccessible subsets are useless)

Using F_A requires a **fast inclusion test** to answer the decision problem:

“is subset $x^{(t)}$ acceptable?” ($x^{(t)} \in F_A?$)

or at least a **fast update test**: if $x^{(t)} \in F_A$, is $x^{(t)} \cup \{i\} \in F_A?$

$F_A = \{x' \subseteq B : \exists x \in X : x' \subseteq x\}$ (subsets of feasible solutions, i.e. **partial solutions**) is a natural candidate, but its **inclusion test**

“is subset $x^{(t)}$ a partial solution?” ($\exists x \in X : x^{(t)} \subseteq x?$)

generalizes the feasibility problem ($\exists x \in X : \emptyset \subseteq x?$)

“is there any feasible solution?” ($\exists x \in X?$)

and could be NP-complete. In that case, one needs to **relax the search space**; include other subsets which may lead to more sub-optimal solutions, while making the acceptability test easier.

Including a partial solution is like asking “is there a solution that includes this subset?” and could be an NP-complete problem, not always polynomial.

If the **objective function** can be extended from X to F_A , it looks natural to use the objective function as the **selection criteria**

$$\phi_A(i, x) = f(x \cup \{i\})$$

Examples of search spaces: The search space F_A may include:

- All feasible solutions

$$F_A = X$$

In the KP, all subsets of feasible weight.

- All partial (and all feasible) solutions

$$F_A = \bigcup_{x \in X} 2^x$$

In the MDP, all subsets of at most k points.

In Kruskal's algorithm for the MSTP, all spanning forests.

- Only some special promising partial solutions

$$F_A \subset \bigcup_{x \in X} 2^x$$

In Prim's algorithm for the MSTP, all trees spanning a given vertex.

- Some special subsets not sufficient to guarantee feasibility

$$F_A \supset \bigcup_{x \in X} 2^x$$

In the TSP, all subsets of arcs not including branching and sub-tours.

4.2 Effectiveness and Efficiency

A constructive algorithm A finds the optimum when at **each step** t the current subset $x^{(t)}$ **is included** in at least one **optimal solution**. An exact algorithm always keeps an open way to an optimal solution.

This property holds in $x^{(0)} = \emptyset$, but is usually lost at some step t ,

A constructive algorithm performs at **most** $n = |B|$ **steps** (can't add more element than the whole ground set) consisting of

1. the **construction** of $\Delta_A^+(x)$
2. the **evaluation** of $\phi_A(i, x)$ for each $i \in \Delta_A^+(x)$
3. the **selection** of the element i minimizing $\phi_A(i, x)$
4. the **update** of x (and auxiliary data structures)

In general, the **complexity** is a **polynomial** of rather low order, dominated by the first two components

$$T_A(n) \in O\left(n \left(T_{\Delta_A^+}(n) + T_{\phi_A}(n)\right)\right)$$

4.3 General features

Constructive algorithms

- are **intuitive**
- are **simple** to design, analyze and implement
- are **very efficient** (usually low complexity)
- have a **strongly variable effectiveness**; they can range from providing an optimal solution on some problem to not even guaranteeing a feasible solution; on most problems they provide solutions of extremely variable quality, often low

It is fundamental to **study the problem** before the algorithm.

When are they used? Constructive algorithms are used

- when they **provide the optimal solution**
- when the **execution time** must be **very short**
- when the **problem** has a **huge size** or requires heavy computation
- as a **component for other algorithms**, for example as
 - **starting phase** for exchange algorithms
 - **basic procedure** for recombination algorithms

Remarkable case: A particularly remarkable case is that in which

- the **objective function** is **extended** from X to F_A
- the **selection criteria is the objective function**

$$\phi_A(i, x) = f(x \cup \{i\})$$

This means that instead of minimizing the selection criteria (which is generic) I use the value of the objective function. It's a greedy algorithm, sometimes can be a good idea.

4.4 Examples

4.4.1 The Fractional Knapsack Problem (FKP)

Select from a set of **objects of identical volume** a **maximum value subset** which could be contained in a **knapsack of limited capacity** (same as the KP but all the objects have the same volume).

In the FKP the capacity simply imposes a **cardinality constraint**: the feasible solutions are those with $|x| \leq \lfloor V/v \rfloor$

Algorithm 1 Algorithm *GreedyFKP(i)*

```

 $x := \emptyset; x^* := \emptyset;$ 
if  $x \in X$  then
     $f^* := f(x);$ 
else
     $f^* := +\infty;$ 
end if
while  $|x| < \lfloor V/v \rfloor$  do
     $i := \arg \max_{i \in B \setminus x} \phi_i;$ 
     $x := x \cup \{i\};$ 
    if  $x \in X$  and  $f(x) > f^*$  then
         $x^* := x; f^* := f(x);$ 
    end if
end while
return  $(x^*, f^*)$ 

```

- Define $F_A = X$: subset x can be extended as long as $|x| < \lfloor V/v \rfloor$
- Any or no element of $B \setminus x$ extends x feasibly ($\Delta_A(x) = B \setminus x$ or \emptyset)
- The objective function is additive, and therefore

$$f(x \cup \{i\}) = f(x) + \phi_i \implies \arg \max_{i \in B \setminus x} f(x \cup \{i\}) = \arg \max_{i \in B \setminus x} \phi_i$$

- The last subset visited is the best solution found

Just add objects until the cardinality is reached, add the most valuable one each time. Each solution is better than the previous one, since it's larger.

The algorithm always finds the optimal solution.

4.4.2 The Knapsack Problem

Select from a set of objects of **different volume** a maximum value subset which could be contained in a knapsack of limited capacity.

The algorithm is similar, the **differences** are:

- The set of elements that can be added is the subset of items which do not exceed the residual capacity
- The termination condition must check that there is at least one element that can extend the current solution

Algorithm 2 Algorithm *GreedyKP(i)*

```

 $x := \emptyset; x^* := \emptyset; f^* := 0$ 
while  $\exists i \in B \setminus x : v_i \leq V - \sum_{j \in x} v_j$  do
     $i := \arg \max_{i \in B \setminus x: v_i \leq V - \sum_{j \in x} v_j} \phi_i$ 
     $x := x \cup \{i\}$ 
end while
return  $(x, f(x))$ 

```

Define $F_A = X$: only some elements of $B \setminus x$ extend x feasibly

$$\Delta_A^+(x) = \left\{ i \in B \setminus x : \sum_{j \in x} v_j + v_i \leq V \right\}$$

The objective function is additive, and therefore

$$f(x \cup \{i\}) = f(x) + \phi_i \implies \arg \max_{i \in \Delta_A^+(x)} f(x \cup \{i\}) = \arg \max_{i \in \Delta_A^+(x)} \phi_i$$

The last subset visited is the best solution found.

4.4.3 Traveling Salesman Problem

Given a directed graph and a cost function defined on the arcs, find a **minimum cost circuit visiting all the nodes** of the graph.

Define F_A as the collection of all subsets of arcs that form no sub-tour and keep a degree ≤ 1 in all nodes, (it is a superset of the partial solutions). The constructive algorithm add each time the cheapest arc among those in F_A , until nothing can be added, that's the solution.

The selection criteria is the objective function (it is additive, therefore equivalent to the cost of the new arc).

Algorithm 3 Algorithm *GreedyTSP(i)*

```

 $x := \emptyset; x^* := \emptyset;$ 
 $f^* := +\infty;$ 
while  $\Delta_A^+(x) \neq \emptyset$  do
     $i := \arg \min_{i \in \Delta_A^+(x)} c_i;$ 
     $x := x \cup \{i\};$ 
end while
if  $x \in X$  then
     $x^* := x; f^* := f(x);$ 
end if
return  $(x^*, f^*)$ 

```

Only the last subset visited can be feasible (if any!).

4.4.4 Maximum Diversity Problem

Select from a set of points a subset of k points with the **maximum sum of the pairwise distances**.

Algorithm 4 Algorithm *GreedyMDP(i)*

```

 $x := \emptyset$ 
while  $|x| < k$  do
     $i := \arg \max_{i \in B \setminus x} \sum_{j \in x} d_{ij}$ 
     $x := x \cup \{i\}$ 
end while
return  $(x, f(x))$ 

```

Define F_A as the set of all partial solutions.

The subset x can be extended as long as $|x| < k$.

Any element of $B \setminus x$ extends x in a feasible way.

The objective function is quadratic, and therefore

$$f(x \cup \{i\}) = f(x) + 2 \sum_{j \in x} d_{ij} + d_{ii} \implies \arg \max_{i \in B \setminus x} \sum_{j \in x} d_{ij}$$

The last subset visited is the best (and only feasible) solution found.

4.4.5 Summary

A constructive heuristic A finds

- The **optimum** when $\Delta_A^+(x^{(t)})$ and $\phi_A(i, x)$ guarantee that the current subset $x^{(t)}$ is **always included** in at least one **optimal solution**.
- A **feasible solution** when $\Delta_A^+(x^{(t)})$ guarantees that the current subset $x^{(t)}$ is **always included** in at least one **feasible solution**.
- A **general subset** when these **properties are lost** at some step t

An ideal constructive algorithm always keeps one open way to the optimal solution.

In practice, some of these properties is usually lost at some step of the algorithm.

4.4.6 Relevant Features

None of the above algorithms where able to find the optimum (except the FKP, examples not provided, but trust me).

What features allow a constructive algorithm to **find the optimum**?

- A **search space identical to the feasible region** ($F = X$)? (No, because this holds for both the fractional and general KP)
- A **cardinality-constrained problem**? (It would explain failing on the KP, but not on the MDP and TSP)
- An **additive objective function**? (It does not explain failing on the TSP)

There is **no general characterization** of the problems solved exactly by constructive algorithms.

But there are characterizations for wide classes of problems.

4.5 The additive case

Assume that

1. the **objective function** is **additive**

$$\exists \phi : B \rightarrow \mathbb{N} : f(x) = \sum_{i \in x} \phi_i$$

2. the **solutions** are the **bases** (maximal subsets) of the **search space**

$$X = B_F = \{Y \in F : \nexists Y' \in F : Y \subset Y'\}$$

The maximal subsets of the search space F , there are no other subsets Y' such that Y is fully included in Y' (can't be bigger).

It is a very frequent case (KP, MAX-SAT, TSP, but not MDP, SCP).

In this case, the constructive algorithm always finds the optimal solution if and only if (B, F) is a **matroid embedding**.

Since the definition of matroid embedding is rather complex, let us focus on some important structures

- greedoids (necessary condition)
- matroids or greedoids with the strong exchange property (sufficient)

4.5.1 Greedoids

A **greedoid** (B, F) (ground set, search space) with $F \subseteq 2^B$ is a **pair** such that

- **Trivial axiom:** $\emptyset \in F$.

The empty set is acceptable.

A greedy algorithm starts from the empty set so it must be included.

- **Accessibility axiom:** if $x \in F$ and $x \neq \emptyset$ then $\exists i \in x : x \setminus \{i\} \in F$.

Any acceptable subset can be built adding elements in suitable order.

Putting element in a given order creates a path through the search space.

If you have a not empty subset in the search space, it's always possible to find an element of the subset that can be removed leaving the result in the search space.

- **Exchange axiom:** if $x, y \in F$ with $|x| = |y| + 1$, then $\exists i \in x \setminus y$ such that $y \cup \{i\} \in F$.

Any acceptable subset can be extended with a suitable element of any other acceptable subset of larger cardinality.

Two subsets in the search space, one has one more element than the second (cardinality +1), you can always find at least one element which is only in the larger subset and can be added to the smaller one in a way that results in another valid subset ($\in F$).

The exchange axiom implies that **all bases have the same cardinality**. If you consider two bases and assume them of different cardinality the axiom says that you can always take an element from the larger and put it into the smaller, while remaining in the search space. This would negate the axiom.

All of these conditions

- hold in the fractional KP, MST problem (both Kruskal and Prim), ...
- do not hold in the general KP, TSP, ...
- hold in the MDP, but the objective function is not additive

Greedoids **make greedy algorithms possible**, but not necessarily exact.

4.5.2 Matroids

A matroid is a **set system** (B, F) with $F \subseteq 2^B$ such that

- **Trivial axiom:** $\emptyset \in F$.

- **Heredity axiom:** if $x \in F$ and $y \subset x$ then $y \in F$.

Any acceptable subset can be built adding its elements in any order.

For any subset inside the search space, its subsets are also inside the search space.

- **Exchange axiom:** if $x, y \in F$ with $|x| = |y| + 1$, then $\exists i \in x \setminus y$ such that $y \cup \{i\} \in F$.

Any acceptable subset can be extended with a suitable element of any other subset of larger cardinality.

The heredity axiom is a **stronger version of accessibility**

- it holds in Kruskal's search space for the MST problem
- it does not hold in Prim's search space for the MST problem

Uniform matroid: fractional and general knapsack

$$F = \{x \subseteq B : |x| \leq \lfloor V/v \rfloor\}$$

- Trivial axiom: the empty set respects the cardinality constraint
- Heredity axiom: if x respects the cardinality constraint, all of its subsets also respect it
- Exchange axiom: if x and y respect the cardinality constraint and $|x| = |y| + 1$, one can always add a suitable element of x to y without violating the cardinality (in fact, any element of x)

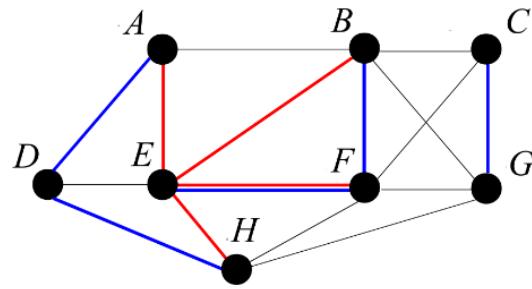
For the general KP the first two axioms hold, but the third one does not.

Example: if $V = 6$ and $v = [3 3 2 2 1]$, the subsets $x = \{3, 4, 5\}$ and $y = \{1, 2\}$ are in F , but no element of x can be added to y .

Graphic matroid: minimum spanning tree

$$F = \{x \subseteq B : x \text{ forms no cycles}\}$$

- Trivial axiom: the empty set of edges forms no cycles
- Heredity axiom: if x is acyclic, all of its subsets are acyclic
- Exchange axiom: if x and y are acyclic and $|x| = |y| + 1$, one can always add a suitable edge of x to y without forming any cycle (not all edges of x work)



$$x = \{(A, D), (D, H), (E, F), (B, F), (C, G)\}$$

$$y = \{(A, E), (B, E), (E, F), (E, H)\}$$

$(A, D), (D, H)$ and (C, G) can be added to y .

TSP: the first two axioms hold

- the empty set has no sub-tours and degrees ≤ 1
- any proper subset of a set $\in F$ (no sub-tours and degrees ≤ 1) also belongs to F

but the third axiom is violated.

Example: $y = \{(1, 2), (2, 3)\}$ and $x = \{(3, 1), (1, 4), (4, 2)\}$.

No arc of x can be added to y remaining in F .

4.5.3 Greedoids with the strong exchange axiom

The optimality of the greedy algorithm can be proved for greedoids (weaker second axiom) if the exchange axiom is strengthened.

Strong exchange axiom:

$$\left\{ \begin{array}{ll} x \in F, y \in B_F & \text{such that } x \subset y \\ i \in B \setminus y & \text{such that } x \cup \{i\} \in F \end{array} \right. \implies \exists j \in y \setminus x : \left\{ \begin{array}{l} x \cup \{j\} \in F \\ y \cup \{i\} \setminus \{j\} \in F \end{array} \right.$$

Given a basis and one of its subsets (from which the basis is accessible), if there is an element that “leads astray” the subset from the base, there must be another one which keeps it on the right way and it must be feasible to exchange the two elements in the basis.

Given a base y and a subspace of the search space x , which is also a subset of the base, you can have an element i not in y that could be feasibly added to x . There must be another element inside y , but not x , that could be added to x and replaced with i in the base, all while remaining in the search space.

MST: A classical example of greedoid with strong exchange axiom is given by

- B = edge set of a graph
- F = collection of the trees including a given vertex v_1

that yields Prim’s algorithm for the MST problem.

The trivial and the accessibility axiom hold (the heredity one does not). The exchange axiom holds in the strong form.

Optimal constructive algorithms

Notice that the optimality of a constructive algorithm A depends on

- the **properties of the problem** (e.g., additive objective function, bases as feasible solutions)
- the **properties of the search space F_A** (that is, of the algorithm)

4.5.4 What to do when the axioms are violated

If a search space **violates** the desired **axioms**, one can try and **change** it.

For the TSP, an alternative F_A includes all paths starting from node 1.

Let N_x be the set of nodes visited from x : the acceptable extensions are all arcs going out of the last node of path x and not closing a sub-tour

$$\Delta_A^+(x) = \{(h, k) \in A : h = \text{Last}(x), k \notin N_x \text{ or } k = 1 \text{ and } N_x = N\}$$

Unfortunately, the axioms are still not all satisfied

- the **trivial** axiom always **holds**
- the **accessibility** axiom **holds**: removing the last arc yields a path starting from node 1
- the **heredity** axiom does **not hold**: not all subsets are paths
- the **exchange** axiom does **not hold**

Therefore, it is **not even a greedoid**.

But the algorithm can still be a **reasonable heuristic**.

The Nearest Neighbour heuristic for the TSP: The Nearest Neighbour (NN) heuristic adopts the **alternative search space** keeping the **objective function** as the **selection criteria**

1. **Start** with an **empty set of arcs**: $x^{(0)} = \emptyset$ that represents a degenerate path going out of node 1 (the optimal solution certainly visits node 1).
2. **Find the arc of minimum cost** going out of the last node of x

$$(i, j) = \arg \min_{(h, k \in \Delta_A^+(x))} c_{hk}$$

(the objective function is additive). It's selecting the "nearest neighbour", i.e. the next closest node, without closing a sub-tour (only unseen nodes).

3. If $j \neq 1$, go back to point 2; otherwise, terminate ($\Delta_A^+(x)$ allows the return to node 1 only at the last step).

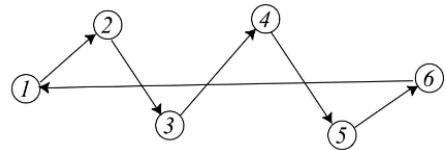
The algorithm is very intuitive, and its **complexity** is $\Theta(n^2)$ (n iterations, n time to choose the arc of minimum cost).

It is not exact, but $\log n$ -approximated (under the triangle inequality). You're going to find $\log n$ times the optimum, where n is the number of nodes in the graph; it's not α approximated.

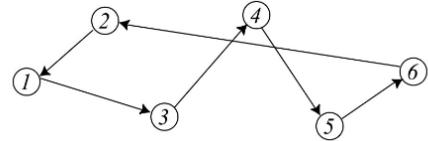
Example: Consider a complete graph (the arcs are not reported for clarity)



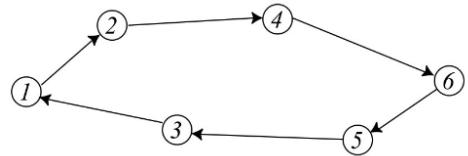
Starting from node 1



Starting from node 2



The optimal solution cannot be found starting from any node



4.6 Heuristic constructive algorithms (HCA): the KP

If the problem does **not admit** a search space with **suitable properties**, one must keep into account the **constraints of the problem** adopting

1. not only a **good definition** of F_A
2. but also a **sophisticated definition of the selection criteria** $\varphi_A(i, x)$

This allows effective results, even if not provably optimal.

In the KP, the drawback derives from the volume of the objects: promising objects have a large value, but also a small volume

- define the **selection criteria** as the unitary value $\varphi_A(i, x) = \phi_i/v_i$ (getting the “value” of each piece to determine which is best)

The resulting algorithm

- can perform very badly
- with a small modification is 2-approximated

Example:

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1
ϕ/v	1.40	0.67	2.00	1.67	4	1

$$V = 8$$

The algorithm performs the **following steps**:

1. $x := \emptyset$
2. select $i := e$ and update $x := \{e\}$
3. select $i := c$ and update $x := \{c, e\}$
4. select $i := d$ and update $x := \{c, d, e\}$
5. select $i := f$ and update $x := \{c, d, e, f\}$ (object a does not fit)
6. since $\Delta_A^+(x) = \emptyset$, terminate

The value of the solution found is 14, the optimal solution is $x^* = \{a, c, e\}$ and its value is 15. There can be critical cases with indefinitely bad gaps (it can discard large objects, even if they have large value).

4.6.1 2-Approximated algorithm for the KP

Steps

1. Start with an **empty subset**: $x^{(0)} = \emptyset$
2. Find the **object $i^{(t)}$ of maximum unitary value** in $B \setminus x^{(t-1)}$:

$$i^{(t)} := \arg \max_{i \in B \setminus x^{(t-1)}} \frac{\phi_{i^{(t)}}}{v_{i^{(t)}}}$$

it considers all elements, not only the one that respect the capacity.

3. If it respects the capacity, **add $i^{(t)}$ to $x^{(t-1)}$** : $x^{(t)} := x^{(t-1)} \cup \{i^{(t)}\}$ and go **back to point 2**.
4. Build a **solution** with the **first rejected object**: $x' = \{i^{(t)}\}$.
5. **Return the better solution** between x and x' : $f_A = \max [f(x), f(x')]$.

It is easy to prove that

- **the sum of the two solution values overestimates the optimum**

$$f(x) + f(x') = \sum_{\tau=1}^t \phi_{i^{(\tau)}} \geq f^*$$

This sum essentially represents a solution for a “bigger knapsack” (it exceeds the capacity), so it will obviously be better (upper bound).

- **the best of the two solution values is at least half their sum**

$$f_A = \max [f(x), f(x')] \geq \frac{f(x) + f(x')}{2} \geq \frac{1}{2} f^*$$

The larger one will obviously be more than half our earlier sum, which is larger than the optimal solution \Rightarrow it will be larger than half the optimal solution.

4.7 Pure and adaptive constructive algorithms

A constructive algorithm A is

- **Pure** if the selection criteria φ_A depends only on the new element i .
The decision is made only on what to add, independently of the “past choices”.
- **Adaptive** if φ_A depends both on i and on the current solution x .

Many criteria $\varphi_A(i, x)$ admit **equivalent forms** depending only on i

- In the TSP, $\varphi_A((i, j), x) = f(x \cup \{(i, j)\})$ is equivalent to c_{ij} .
- In the KP, $\varphi_A(i, x) = f(x \cup \{i\})$ is equivalent to ϕ_i

So far, we have seen only pure constructive algorithms.

An **additive selection criteria** yields a **pure constructive algorithm**.

4.8 HCA: Set Covering

Given a binary matrix and a cost vector associated to the columns, find a minimum cost subset of columns covering all the rows (at least one 1 for each row, choose columns, minimum cost).

The **objective** is **additive**, but the solutions are **not maximal subsets** (actually, the smaller feasible subsets are better).

An **adaptive selection criteria** $\varphi_A(i, x)$ is necessary: a pure one ($\varphi_A(i)$) could repeatedly choose columns covering the same rows.

The more promising **ideas** are to consider

- the **objective function**: select columns of low cost
- the **constraints**: select columns covering many rows
- the **current subset** x : select columns covering new rows

In **summary**

- include in $\Delta_A^+(x)$ only **columns covering additional rows** not in x
- apply the **adaptive selection criteria** $\varphi_A(i, x) = c_i/a_i(x)$ where $a_i(x)$ is the number of rows covered by i , but not by x .

In the calculation we must take into account the columns we already covered, the selection criteria must consider only columns that have yet to be covered. This means that the algorithm must be adaptive, it can't be pure, otherwise it would choose "useless" columns.

This is taken into account inside the definition of $a_i(x)$, which is defined as the number of rows covered by i , but not by x .

Example: this algorithm has the optimum as an upper bound, but it can also fail

$$c \quad \boxed{25 \ 6 \ 8 \ 24 \ 12}$$

$$A \quad \begin{array}{|ccccc} \hline 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

The algorithm performs the following **steps**:

- $x := \emptyset$
- since $c/a_i(x) = [4.1\bar{6}, 2, 4, 12, 12]$, select $i := 2$
- since $c/a_i(x) = [8.\bar{3}, -, 8, 12, 12]$, select $i := 3$
- since $c/a_i(x) = [12.5, -, -, 24, 12]$, select $i := 5$
- since $c/a_i(x) = [25, -, -, 24, -]$, select $i := 4$
- all the rows are covered, therefore $\Delta_A^+(x) = \emptyset$ and terminate

The solution returned is $x = \{2, 3, 4, 5\}$ and its value is 50, whereas the optimal solution $x^* = \{1\}$ has value $f^* = 25$.

4.8.1 Approximability of the SCP

This algorithm has a **non-constant** (logarithmic) **approximation ratio**

- At each step t , each column i is evaluated with **criteria**

$$\varphi_A(i, x^{(t-1)}) = \frac{c_i}{a_i(x^{(t-1)})}$$

estimate of the unitary cost, cost of a column divided by the number of covered rows that are not already covered by the current solution.

- **Row j is covered** by a certain **column** (i_j) at a certain **step** (t_j).
- Start assigning **weight** $\theta_j = 0$ to each **row j** .
- When each **row j is covered** (step t_j), set its **weight** to

$$\theta_j = \frac{c_{i_j}}{a_{i_j}(x^{(t_j-1)})}$$

so that the total weight of the rows increases by c_{i_j} at step t_j ; correspondingly, x includes column i_j and its cost increases by c_{i_j} .

When you choose a column, put it in the partial solution and increase the cost of the solution by c_{i_j} , now modify the weights of the rows, take the cost of the column c_{i_j} and divide it by how many new rows are covered, give each of these rows this increase in weight.

- The **total cost** of x is always equal to the **total weight of the rows**

$$f_A(x) = \sum_{i \in x} c_i = \sum_{j \in R} \theta_j$$

- at **step t** , there are $|R^{(t)}| \leq |R| - t$ **uncovered rows**.

- the columns of the optimal solution could cover them all with **cost** f^* \implies at least one of such columns has **unitary cost** $\leq f^*/|R^{(t)}|$.
- the column i selected has **minimum unitary cost** $\varphi_A(i, x^{(t-1)})$, therefore $\leq f^*/|R^{(t)}|$ and the covered rows **increase their weight** by

$$\theta_j = \varphi_A(i, x^{(t-1)}) \leq \frac{f^*}{|R^{(t_j)}|} \implies \sum_{j \in R} \theta_j \leq \sum_{j \in R} \frac{f^*}{|R^{(t_j)}|}$$

The cost to cover each row j is not larger than the optimum divided by the number of rows uncovered at the step in which j gets covered.

- the integer number $|R^{(t)}|$ strictly **decreases** at each step.
- the sum can be **overestimated reducing** $|R^{(t)}|$ by 1 at each step.
- The **approximation ratio** is limited by a **logarithmic guarantee**

$$f_A = \sum_{j \in R} \theta_j \leq \sum_{j \in R} \frac{f^*}{|R^{(t_j)}|} \leq \sum_{r=|R|}^1 \frac{f^*}{r} \leq (\ln |R| + 1) f^*$$

4.9 HCA: Bin Packing Problem

Another adaptive problem, the BPP requires to divide a set O of voluminous objects into the minimum number of containers of given capacity drawn from a set C .

$B = O \times C$ includes the **object-container assignments** (i, j)

- with exactly **one container for each object**
- with the **total volume** in each container **not exceeding the capacity**

Let us define the **search space** F_A as the set of **all partial solutions**.

The **objective function** is a **bad selection criteria**, because it is flat.
All the augmented subsets have the same value or increase it by 1.

4.9.1 First-Fit heuristic

Consider the **object-container pairs** lexicographically

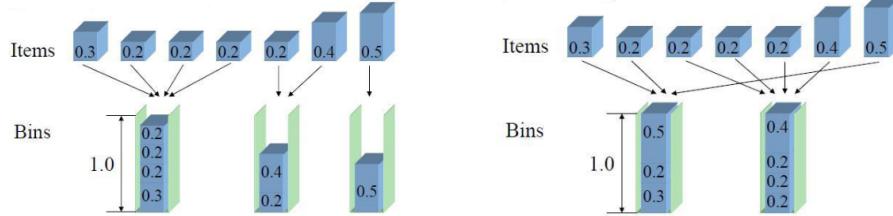
- **Start** with an **empty subset**: $x^{(0)} = \emptyset$
- **Select pair** (i, j) according to the following criteria:
 - i is the **first** (minimum index) **unassigned object**
 - j is the **first container with enough residual capacity** for i
(a used container, if possible; an unused one otherwise)
- **Add** the new **assignment** to the **solution**: $x^{(t)} := x^{(t-1)} \cup \{(i, j)\}$

Notice that the **choice** of (i, j)

- **does not minimize** $f(x \cup \{(i, j)\})$ (another i could be better)
- is split into **two phases** (first i , then j)

Essentially, take each element and put it into the first container you find with enough space, used or otherwise.

Example:



The solution is not optimal; but it is approximated:

- At least $f^* \geq \sum_{i \in O} v_i / V$ containers are necessary. Lower bound, the minimum number of container if you could completely fill them all, it must be \leq than the optimal solution.
- The occupied volume is $> V/2$ for all used containers, possibly except for the last one (if a second half-empty container existed, its objects would have been assigned to the first, it would be a contradiction).
- The total volume exceeds that of the $f_A - 1$ “saturated” containers

$$\sum_{i \in O} v_i > (f_A - 1) \frac{V}{2}$$

This means that the total volume is more than $f_A - 1$ (if the last is not more than half full) times $V/2$ (lower bound).

Which implies $(f_A - 1) < 2/V \sum_{i \in O} v_i \leq 2f^* \Rightarrow f_A \leq 2f^*$. Our solution is less than the volume of the “saturated” containers expressed before, which in turn is smaller than the optimal solution, refactor the equation and we get the last part, our solution is less than twice the optimum +1.

The analysis can be improved to 1.7.

Better choices? The **approximation ratio** $\alpha = 2$ holds for **any permutation** of the objects.

Intuition would suggest selecting first the smallest objects, in order to keep the objective $f(x \cup \{i\})$ as small as possible, but this neglects that all objects must be assigned.

By contrast, it is **better to select the largest object first** because

- each object in a container has a volume strictly larger than the residual capacity of all the previous containers (otherwise, it would have been assigned to one of them)
- keeping the smallest objects in the end guarantees that many containers have a small residual capacity

This algorithm has a **better approximation ratio**: $f_A \leq (11/9)f^* + 1$.

The difference is that instead of choosing the object of minimum index you choose the object of largest volume.

4.10 Extensions of the basic constructive scheme

The **basic scheme** of constructive algorithms can be **enhanced** using

1. a **more effective construction graph**
 - **add more than one element** to the current subset x
 - add elements to x , but **also remove elements** from x (without going back to visited subsets, otherwise the construction graph would be cyclic)
2. a **more sophisticated selection criteria**, such as
 - a **regret-based function** that estimates potential future losses associated with element i
 - a **look-ahead function** that estimates the final value of the objective obtained adding i to x

4.10.1 Extensions of the construction graph

The constructive algorithm adds an element at a time to the solution.

It is possible to **generalize** this scheme with algorithms that at each step

1. **Add more than one element:** the selection criteria $\varphi_A(B^+, x)$ identifies a subset $B^+ \subseteq B \setminus x$ to add, instead of a single element i .
The criteria depends not only from x but also on the subset of the ground set B^+ .
2. Add elements, but also **remove a smaller number of elements:** the selection criteria $\varphi_A(B^+, B^-, x)$ identifies a subset $B^+ \subseteq B \setminus x$ to add and a subset $B^- \subseteq x$ to remove, with $|B^+| > |B^-|$.
The criteria becomes a function of x , B^+ and the subsets of element currently included in x to remove B^- .

These algorithms build an **acyclic construction graph** on the search space, so that they never revisit any subset.

The fundamental problem is to **define a family** $\Delta_A^+(x)$ of subset pairs such that **optimizing the selection criteria is a polynomial problem**

$$\min_{(B^+, B^-) \in \Delta_A^+(x)} \varphi_A(B^+, B^-, x)$$

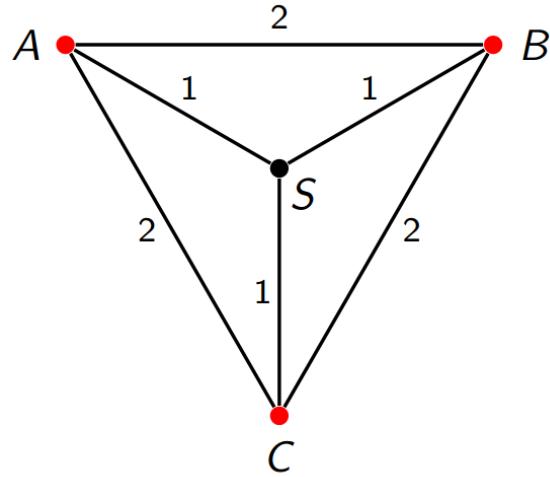
that is

- subsets **efficiently optimizable** (minimum paths, ...)
- subsets of **limited size** (e.g., $|B^+| = 2$ and $|B^-| = 1$)

Minimizing the selection criteria can become difficult, there could be exponentially many subsets. To counteract that we can limit the size of our subsets B^+ and B^- or define specific families for these subsets such that they are “classic problems” and consequently easily optimizable (e.g. minimum paths).

4.11 The Steiner Tree Problem (STP)

Given an undirected graph $G = (V, E)$, a cost function $c : E \rightarrow \mathbb{N}$ on the edges and a subset of special vertices $U \subset V$, find a tree connecting at minimum cost all special vertices.



The minimum tree spanning the special vertices is not necessarily optimal (and it might not even exist).

Basically, connect the special dots and make it quick.

4.11.1 Distance Heuristic (DH) for the STP

A basic constructive algorithm could adopt the same **search spaces** as

- Kruskal's algorithm: the set of **all forests**
- Prim's algorithm: the set of **all trees including a (special) vertex**

but adding **one edge at a time**

- returns solutions with **redundant edges**, therefore expensive
- has a hard time **distinguishing useful and redundant edges**

The Distance Heuristic adopts as **search space F** the **collection of all trees including a given special vertex v_1** (as in Prim).

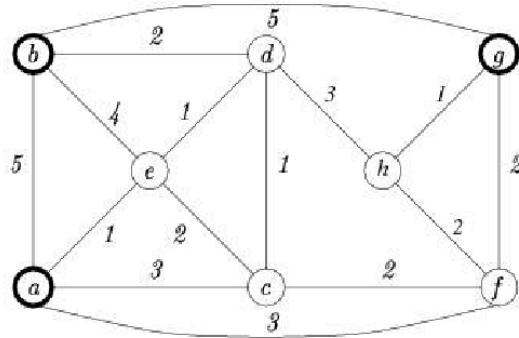
It iteratively **adds a path B^+ between x and a special vertex** instead of a single edge, so that

- x remains a tree
- x spans a new special vertex
- the **minimum cost path** can be **computed efficiently** at each step

It **terminates** when all special vertices are connected.

Instead of adding only one edge at a time, at each step it adds the minimum cost path from an already visited vertex (not necessarily special) to the next closest special vertex, keeping the solution less redundant and simplifying the computation of the minimum cost path.

Example:



- Start with a single special vertex a : $x := \emptyset$ (degenerate tree).
- Add the closest special vertex (b) through path (a, e, d, b) :

$$x = \{(a, e), (e, d), (d, b)\}$$

- Add the closest special vertex (g) through path (g, h, d) :

$$x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$$

The Distance Heuristic algorithm is 2-approximated: it is equivalent to computing a minimum spanning tree on a graph with

- vertices reduced to the special vertices
- edges corresponding to the minimum paths

You can think of it as a graph with only special vertices and the minimum path between them as edges.

- All special vertices are in the solution: terminate (this time the solution is optimal)

Counterexample to optimality: Consider a complete graph $G = (V, E)$ with $U = V \setminus \{1\}$ and cost

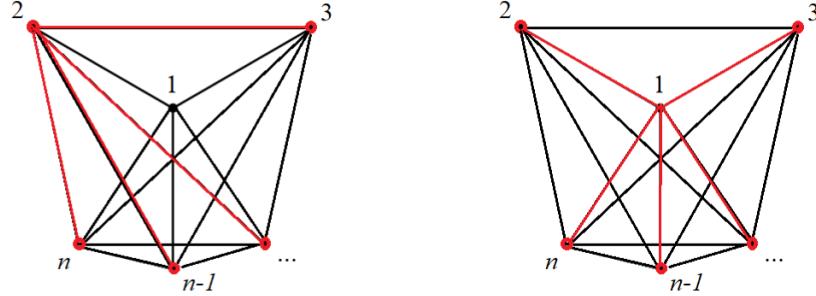
$$c_{uv} = \begin{cases} (1 + \epsilon)M & \text{for } u \text{ or } v = 1 \\ 2M & \text{for } u, v \in U \end{cases}$$

(M is just used to obtain integer costs for any ϵ).

Every arc from node 1 cost $(1 + \epsilon)M$, $2M$ for everything else.

The DH returns a star spanning the special vertices: $f_{DH} = (n - 2) \cdot 2M$.

The optimal solution is a spanning star centered in 1: $f^* = (n - 1) \cdot (1 + \epsilon)M$.



The approximation ratio is

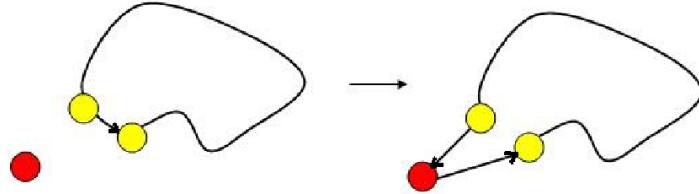
$$\rho_{DH} = \frac{f_{DH}}{f^*} = \frac{n - 2}{n - 1} \cdot \frac{2}{1 + \epsilon} < 2$$

and converges to 2 as n increases and ϵ decreases.

4.12 Insertion algorithms for the TSP

Several heuristic algorithms for the TSP define the **search space** F_A as the set of all circuits of the graph including a given node; a circuit

- cannot be obtained from another one by adding a single arc
- can be obtained adding two arcs (i, k) , (k, j) and removing one (i, j)



Steps

1. Start with a **zero-cost self-loop on node 1**: $x^{(0)} = \{(1, 1)\}$ (Not much different from an **empty set**)
2. Select a **node k** to be **added** and an **arc (i, j)** to be **removed**
3. If the circuit **does not visit all nodes**, go back to point 2; **otherwise terminate**

Such a scheme **never visits again the same solution** and builds a **feasible solution in $n - 1$ steps** (each step adds a new node).

Start from a circuit, add one node at a time by adding a new edge and removing an old one.

4.12.1 Cheapest Insertion heuristic

The selection criteria $\varphi_A(B^+, B^-, x)$ must **choose an arc and a node**; there are $(n - |x|)|x| \in O(n^2)$ alternatives

- $|x|$ possible **arcs** (s_i, s_{i+1}) to **remove**; the nodes already in the circuit
- $n - |x|$ possible **nodes** k to **add** through the arcs (s_i, k) and (k, s_{i+1}) ; nodes not yet in the circuit that can be added

The **Cheapest Insertion** (CI) heuristic uses as a **selection criteria**

$$\varphi_A(B^+, B^-, x) = f(x \cup B^+ \setminus B^-)$$

Objective function $f(x)$ is **additive**, hence **extensible** to the whole of F_A .

Here k denotes a general node outside the circuit, s_i (or anything s -related) denotes nodes already inside the circuit.

Since $f(x \cup B^+ \setminus B^-) = f(x) + c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}}$ (objective function + the cost of the arcs added – the cost of the arc removed)

$$\arg \min_{B^+, B^-} \varphi_A(B^+, B^-, x) = \arg \min_{i, k} (c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$$

We need to minimize these terms by choosing the set of i, k that minimizes the variation in cost.

The **computational cost of evaluating** φ_A **decreases** from $\Theta(n)$ to $\Theta(1)$ (since we need to compute only the variation).

Algorithm Cheapest Insertion:

1. Start with a **zero-cost self-loop** on node 1: $x^{(0)} = \{(1, 1)\}$. It is also like starting with a single node.
2. Select the **arc** $(s_i, s_{i+1}) \in x$ and the **node** $k \notin N_x$ **such that** $(c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$ is minimum; select the node to add and the arc to remove, such that the total variation is at a minimum.
3. If the circuit does not visit all nodes, go back to point 2; otherwise terminate.

It is not exact, but **2-approximated**, under the triangle inequality.

The CI algorithm performs $n - 1$ steps: at each step t

- it evaluates $(n - t)t$ node-arc pairs
 - each evaluation requires constant time
 - each evaluation possibly updates the best move
- it performs the best addition/removal
- it decides whether to terminate

The **overall complexity** is $\Theta(n^3)$.

It can be reduced to $\Theta(n^2 \log n)$ collecting in a min-heap the insertion costs for each external node: each of the n steps

- selects the best insertion in $O(n)$ time and performs it
- creates two new insertions and removes one for each external node, and updates their heaps in $O(n \log n)$ time

4.12.2 Nearest Insertion heuristic

The algorithm Cheapest Insertion tends to select nodes close to circuit x : minimizing $c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}}$ implies that $c_{s_i,k}$ and $c_{s_{i+1},k}$ are small (looks for the closest node to add, the weight of the arcs to that node must be small).

To accelerate, one can **decompose criteria** φ_A **into two phases**.

Algorithm Nearest Insertion (NI):

1. start with a **zero-cost self-loop** on node 1: $x^{(0)} = \{(1, 1)\}$
2. **Add criteria:** select the **node k nearest to circuit x**

$$k = \arg \min_{l \notin N_x} \left(\min_{s_i \in N_x} c_{s_i,l} \right)$$

It chooses the minimum distance among the minimum distances of each node from the circuit.

3. **Delete criteria:** select the **arc (s_i, s_{i+1}) that minimizes f**

$$(s_i, s_{i+1}) = \arg \min_{s_i, s_{i+1} \in x} (c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$$

The arc to break is the one that minimizes the objective function.

4. If the circuit does not visit all nodes, go back to point 2; otherwise terminate

It is not exact, but **2-approximated**, under the triangle inequality.

Instead of choosing together node and arc, you first choose the node and, given the node, the arc is chosen; you look for the node closest to the circuit in general.

Instead of minimizing the new cost, the cost of the arc that connects the new node to the circuit is minimized.

The NI algorithm performs $n - 1$ steps: at each step t it

- **Evaluates the distance** of $(n - t)$ nodes from the circuit (the remaining nodes), each one in $\Theta(t)$ time (confront them with the nodes already in the circuit); it makes this $\Theta(n^2)$.
- **Selects the node at minimum distance.**
- **Evaluates the removal** of t arcs, each one in $\Theta(1)$ time.
- Performs the **best addition/removal**.
- **Decides** whether to **terminate**.

The **overall complexity** is $\Theta(n^3)$.

It **can be reduced to** $\Theta(n^2)$ collecting in a vector for each external node the closest internal node: each of the $n - 1$ steps

- selects the closest node in $O(n)$ time
- finds the insertion point in $O(n)$ time
- inserts the node creating a new internal node for each external node, which possibly becomes the closest saved in the vector; each of the $O(n)$ updates takes $O(1)$ time

It has the same approximation guarantee as the last heuristic, but it's faster.

4.12.3 Farthest Insertion heuristic

The choice of the closest node to the cycle is natural, but misleading: since all nodes must be visited, it is preferable to service in the best way the most problematic ones (i.e., the farthest ones).

Algorithm Farthest Insertion (FI):

1. Start with a **zero-cost self-loop** on node 1: $x^{(0)} = \{(1, 1)\}$.
2. **Add criteria:** select the node k **farthest** from cycle x

$$k = \arg \max_{l \notin N_x} \left(\min_{s_i \in N_x} c_{s_i, l} \right)$$

(the node that is farthest from the closest node of the cycle) the maximum of the minimum distances from the circuit.

3. **Delete criteria:** select the arc (s_i, s_{i+1}) minimizing

$$(s_i, s_{i+1}) = \arg \min_{(s_i, s_{i+1}) \in x} (c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$$

4. If the circuit does not visit all nodes, go back to point 2; otherwise terminate.

It is **log n -approximated** under the triangle inequality, hence **worse** than the previous ones in the **worst-case** (but often **experimentally better**).

It starts by inserting the furthest possible nodes (but includes them in the best way), until it forms a complete circuit. This way the “isolated” nodes can be connected first and penalize less the final result.

The FI algorithm performs $n - 1$ steps: at each step t

- it **evaluates the distance** of $(n - t)$ nodes from the circuit, each one in $\Theta(t)$ time
- **select** the node at **maximum distance**
- it **evaluates the removal** of t arcs, each one in $\Theta(1)$ time
- it performs the **best addition/removal**
- it **decides** whether to **terminate**

The complexity is the same as the NI (just with the farthest).

The overall complexity is $\Theta(n^3)$.

It can be reduced to $\Theta(n^2)$ as in the NI heuristic.

4.13 Regret-based constructive heuristics

We've seen how to make a more effective construction graph, the other method to extend a constructive algorithm is a more sophisticated selection criteria, such as a look-ahead or regret-based function.

Decisions taken in early steps can severely restrict the feasible choices in later steps due to the **constraints** of the problem

- **BPP:** all objects must be put into a container, but early assignments could make some containers unavailable for later objects
- **TSP:** all nodes must be visited, but early routing decisions could make the visit of later nodes more expensive (even impossible, if the graph is noncomplete)
- **CMST:** all vertices must be linked to the root through a subtree, but early links could make some subtrees unavailable for later vertices

The selection criteria can take it into account **implicitly**

- **BPP:** the Decreasing First-Fit heuristic assigns the larger objects first
- **TSP:** the Farthest Insertion heuristic visits the farther nodes first

Some selection criteria aim explicitly to **leave larger sets of good choices**.

Regret criteria: A typical regret-based heuristic consists in

- **partitioning** $\Delta_A^+(x)$ into disjoint classes of choices (the assignments of each object, how every object can be assigned, the edges incident in each vertex)
- compute a **basic selection criteria** for all choices
- compute **for each class the regret**, i.e. the **difference** between either
 - the **second-best choice** or
 - the **average of the other choices** (possibly weighted)and the best choice in order to estimate the damage incurred by postponing the best choice until it becomes impossible
- **choose the best choice** of the class for which the **regret is maximum**. This is effective when a single choice per class must be taken

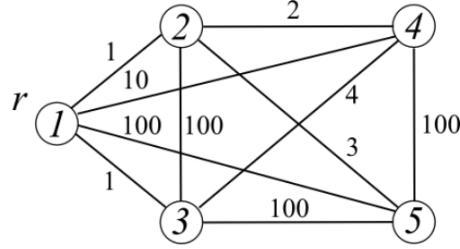
Essentially we divide the possible additions into classes (for example all the feasible assignments of an object is a single class), and pick the (greedy) best

choice for each class. To choose from which class to pick the best choice, we use the regret, maximizing it.

The regret represents how much worse the other choices will be, so we choose the best choice of the class with the highest regret, “fearing” that if we don’t pick that option now, we’ll be forced to pick a much worse one later on.

You can make this reasoning when you’re sure that you will have to take at least one choice for each class.

Example: Consider the CMSTP and ground set $B = V \times T$ ((vertex, subtree) pairs). Let the weights be uniform ($w_v = 1$ for all $v \in V$) and capacity $W = 2$



Let the search space F include all partial solutions.

The greedy algorithm puts vertex 2 in subtree 1, vertex 3 in subtree 2; then vertex 4 in subtree 1 and finally vertex 5 in subtree 3:

$$c(x) = 1 + 1 + 2 + 100 = 104$$

The regret algorithm puts vertex 2 in subtree 1, vertex 3 in subtree 2; now:

- the regret of vertex 3 is the difference $c(3, 3) - c(3, 2) = 1 - 1 = 0$
- the regret of vertex 4 is the difference $c(4, 2) - c(4, 1) = 10 - 2 = 8$
- the regret of vertex 5 is the difference $c(5, 2) - c(5, 1) = 100 - 3 = 97$

The algorithm puts vertex 5 in subtree 1.

Then, it proceeds putting vertices 2 and 4 in subtree 2:

$$c(x) = 1 + 3 + 1 + 4 = 9$$

4.14 Roll-out heuristics

They are also known as **single-step look-ahead constructive heuristics** and were proposed by Bertsekas and Tsitsiklis (1997).

Given a **basic constructive heuristic A**

- **start** with an **empty subset**: $x^{(0)} = \emptyset$
- at each step t
 - **extend the subset in each feasible way**: $x^{(t-1)} \cup \{i\}$, $\forall i \in \Delta_A^+(x)$
 - **apply the basic heuristic to each extended subset** and compute the resulting solution $x_A(x^{(t-1)} \cup \{i\})$
 - use the **value of the solution** as the **selection criteria** to choose $i^{(t)}$
$$\varphi_A(i, x) = f(x_A(x^{(t-1)} \cup \{i\}))$$
- **terminate** when $\Delta_A^+(x)$ is empty

Try every feasible move, look at the result, go back and choose the move.

The **result** of the roll-out heuristic **dominates that of the basic heuristic** (under very general conditions). I'm trying every possibility, of course it's going to be better.

The **complexity remains polynomial**, but is **much larger** (I have to run the algorithm many more times): in the worst case, $T_{ro(A)} = |B|^2 T_A$.

The basic complexity is multiplied by the dimension of the ground set squared, because at each point you have to make several attempts and for each one run the algorithm (“standard” complexity times the number of attempts).

Example: roll-out for the SCP

$$c \quad \boxed{25 \ 6 \ 8 \ 24 \ 12}$$

$$A \quad \begin{array}{ccccc} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array}$$

1. start with the empty subset: $x^{(0)} = \emptyset$
 2. for each column i , apply the constructive heuristic starting from subset $x^{(0)} \cup \{i\} = \{i\}$
 - for $i = 1$, obtain $x_A(\{1\}) = \{1\}$ of cost $f_A(\{1\}) = 25$
 - for $i = 2$, obtain $x_A(\{2\}) = \{2, 3, 5, 4\}$ of cost $f_A(\{2\}) = 50$
 - for $i = 3$, obtain $x_A(\{3\}) = \{3, 2, 5, 4\}$ of cost $f_A(\{3\}) = 50$
 - for $i = 4$, obtain $x_A(\{4\}) = \{4, 2, 5\}$ of cost $f_A(\{4\}) = 43$
 - for $i = 5$, obtain $x_A(\{5\}) = \{5, 2, 3, 4\}$ of cost $f_A(\{5\}) = 50$
- It tries every possible solution (in this case “fixing” a starting column)
3. the best solution is the first one, therefore $i^{(1)} = 1$
 4. all rows are covered: the algorithm terminates

4.14.1 Generalized roll-out heuristics

The scheme can be **generalized**

- applying several basic heuristics $A^{[1]}, \dots, A^{[l]}$
- increasing the number of look-ahead steps, i.e., using $x^{(t-1)} \cup B^+$ with $|B^+| > 1$ (add more than one element and check all possibilities)

The **result improves** and the **complexity worsens** further.

The overall scheme does not change significantly

- start from the empty subset: $x^{(0)} = \emptyset$
- at each step t
 - for each possible extension $B^+ \in \Delta_A^+(x^{(t-1)})$ apply each basic algorithm $A^{[l]}$ starting from $x^{(t-1)} \cup B^+$
 - the selection criteria is $\min_l f_{A^{[l]}}(x^{(t-1)} \cup B^+)$
 - use the value of the best solution as the selection criteria for $i^{(t)}$

$$\varphi_A(i, x) = \min_{L=1, \dots, l} f(x_A(x^{(t-1)} \cup \{i\}))$$

- when $\Delta_A^+(x)$ is empty, terminate

4.15 Destructive heuristics

It is an approach exactly complementary to the constructive one

- **Start** with the **full ground set**: $x^{(0)} := B$.
- **Remove an element at a time**, selected
 - so as to **remain within the search space** F_A

$$\Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in F_A\}$$

- **maximizing the selection criteria** $\varphi_A(i, x)$ (usually a cost reduction)

- **Terminate** when $\Delta_A^+(x) = \emptyset$ (there is no way to remain in F_A)

A destructive heuristic (for a minimization problem) can be described as Optimal for the Minimum Spanning Tree Problem.

Algorithm 5 Algorithm *Stingy*(I)

```

 $x := B; x^* := B$ 
if  $x \in X$  then
     $f^* := f(x)$ 
else
     $f^* := +\infty$ 
end if
while  $\Delta_A^+(x) \neq \emptyset$  do
     $i := \arg \max_{i \in \Delta_A^+(x)} \varphi_A(i, x)$ 
     $x := x \setminus \{i\}$ 
    if  $x \in X$  and  $f(x) < f^*$  then
         $x^* := x$ 
         $f^* := f(x)$ 
    end if
end while
return  $(x^*, f^*)$ 

```

Essentially this is “throwing away” the most expensive edges one at a time, without forming a disconnected graph.

Why are they less used? When the **solutions** are much smaller than the ground set ($|x| \ll |B|$) a destructive heuristic

- requires a **larger number of steps**
- is **more likely to make a wrong decision** at an early step
- sometimes **requires more time to evaluate** $\Delta_A^+(x)$ and $\varphi_A(i, x)$

When a **constructive heuristic returns redundant solutions**, it is useful to **append a destructive heuristic** at its end as a post-processing phase.

This auxiliary destructive heuristic

- **starts from the solution** x of the constructive heuristic, instead of B
- adopts as a **search space** the **feasible region**:

$$F_A = X \implies \Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in X\}$$

- adopts as the **selection criteria** the **objective function**:

$$\varphi_A(i, x) = f(x \setminus \{i\})$$

- **terminates after very few steps**

Constructive/destructive heuristic example for the SCP:

$$c \quad \boxed{6 \quad 8 \quad 24 \quad 12}$$

$$A \quad \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}$$

1. The constructive heuristic selects, in order, columns 1, 2, 4 and 3 (each one covers new rows).
2. The solution is redundant: column 2 can be removed (the following columns also cover already covered rows).
3. The auxiliary destructive heuristic removes column 2 and provides the optimal solution $x^* = \{1, 3, 4\}$ (columns 1, 3 and 4 are essential to cover rows 1, 2, 5 and 6).

Summary about constructive and destructive algorithms

Constructive and destructive algorithms

1. Are **intuitive**.
2. Are **simple to design, analyze and implement**.
3. Are **very efficient** (low-order polynomials)

$$T_A(n) \in O\left(T_{\Delta_A^+}(n) + T_{\varphi_A}(n)\right)$$

where

- $T_{\Delta_A^+}(n)$ is the cost to identify $\Delta_A^+(x)$
- $T_{\varphi_A}(n)$ is the cost to evaluate $\varphi_A(i, x)$ for each $i \in \Delta_A^+(x)$
- the selection of $\arg \min \varphi_A(i, x)$ and update of x (and auxiliary data structures) are dominated

4. Have a **strongly variable effectiveness**
 - on some problems they guarantee an optimal solution
 - on other problems they provide an approximation guarantee
 - on most problems they provide solutions of extremely variable quality, often scarce
 - on some problems they cannot even guarantee a feasible solution

It is fundamental to **study the problem before the algorithm**.

When are they used? Constructive and destructive algorithm are used

1. when they provide the optimal solution
2. when the execution time must be very short (e.g., for on-line problems: schedulers, on-call services, ...)
3. when the problem has a huge size or requires heavy computations (e.g., some data are obtained by simulation)
4. as component of other algorithms, for example as
 - starting phase for exchange algorithms
 - basic procedure for recombination algorithms

5 Constructive Metaheuristics

The constructive algorithms have strong limitations on many problems.
What can be done, without abandoning the general scheme?

Iterate the scheme to generate many (potentially) **different solutions**:

- the **efficiency decreases**: the computational times are summed
- the **effectiveness increases**: the best solution is returned

The trade-off must be carefully tuned.

The iterated scheme can apply

- **multi-start**, that is **different algorithm** at **each iteration** $l = 1, \dots, \ell$ (this requires to define multiple \mathcal{F}_{A_l} and φ_{A_l})

but it is more flexible to apply **metaheuristics**, that exploit

- **randomization** (operations based on a random seed), as in the case of semigreedy algorithms, *GRASP* and *Ant System* (partly, *ART*)
- **memory** (operations based on the solutions of previous iterations), as in the case of *ART*, cost perturbation and *Ant System*

Termination condition: The iterated scheme can ideally proceed for an infinite time.

In practice, one uses termination conditions that can be “absolute”

1. a given **total number of iterations** of the basic scheme
2. a given **total execution time**
3. a given **target value of the objective**

or “relative” to the profile of f^*

1. a given **number of iterations** of the basic scheme **without improving** f^*
2. a given **execution time without improving** f^*
3. a given **minimum ratio between the improvement of** f^* **and the number of iterations** of the basic scheme **or the execution time** (e.g.: f^* improves less than 1% in the last 1000 iterations)

Fair comparisons require absolute conditions.

Multi-start: (or restart) is a classical, very simple and natural approach:

- define **different search spaces** $F_{A^{[l]}}$ and **selection criteria** $\varphi_{A^{[l]}}(i, x)$
- **apply each resulting algorithm** $A^{[l]}$ to obtain the respective solution $x^{[l]}$
- **return the best solution** $x = \arg \min_{l=1, \dots, \ell} f(x^{[l]})$

A typical case is to **tune** $\varphi_A(i, x)$ with numerical parameters μ .

The **construction graph can model this situation** by

- including **all nodes and arcs admitted by at least one algorithm** $A^{[l]}$:

$$\mathcal{F}_A = \bigcup_{l=1}^{\ell} \mathcal{F}_{A^{[l]}}$$

- setting **arc weights depending on** $l : \varphi_A(i, x, l) = \varphi_{A^{[l]}}(i, x)$
- setting an **infinite arc weight** for the **arcs** that are **forbidden in a specific algorithm** $A^{[l]} : \varphi_A(i, x, l) = +\infty$

Essentially, this means defining and running different heuristics one after the other and returning the best solution found.

Example: a family of heuristics for the TSP can be obtained setting:

- **insertion criteria:**

$$i_k^* = \arg \min_{i \in \{1, \dots, |x|\}} \gamma_{i,k} = \mu_1(c_{s_i, k} + c_{k, s_{i+1}}) - (1 - \mu_1)c_{s_i, s_{i+1}}$$

where $\mu_1 \in [0; 1]$ tunes the relative strength of the

- increase in cost due to the added node k
- decrease in cost due to the removed edge (s_i, s_{i+1})

- **selection criteria:**

$$k^* \arg \min_{k \in N \setminus N_x} \varphi_A(k, x) = \mu_2 \gamma_{i_k^*, k} + \mu_3 d(x, k) + (1 - \mu_2 - \mu_3)(-d(x, k))$$

where $\mu_2, \mu_3 \in [0; 1]$ tune the relative strength (and sign) of the

- increase in cost due to the added node k
- distance of the added node k from the current circuit x

This yields CI for $\mu = (1/2, 1, 0)$, NI for $\mu = (1/2, 0, 1)$, FI for $\mu = (1/2, 0, 0)$.

5.1 Constructive metaheuristics

The main constructive metaheuristics are

1. **Adaptive Research Technique (ART) or Tabu Greedy:** forbid some moves based on the solutions of the previous iterations

$$\min_{i \in \Delta_{A^{[l]}}^+(x)} \varphi_A(i, x)$$

$$\text{with } \Delta_{A^{[l]}}^+(x) = \left\{ i \in B \setminus x : x \cup \{i\} \in \mathcal{F}^{[l]}(x_A^{[1]}, \dots, x_A^{[l-1]}) \subseteq \mathcal{F} \right\}$$

This is much less popular than the other two.

A greedy algorithm with some “taboo”, some forbidden choices; same selection criteria but restricted search space based on previous solutions.

2. **Semigreedy and GRASP:** use a randomized selection criteria

$$\min_{i: x \cup \{i\} \in \mathcal{F}} \varphi_A^{[l]}(i, x, \omega^{[l]})$$

The selection criteria has a random element, the selection will be partly stochastic. The search space is fixed.

3. **Ant System (AS):** use a randomized selection criteria depending on the solutions of the previous iterations

$$\min_{i: x \cup \{i\} \in \mathcal{F}} \varphi_A^{[l]}(i, x, \omega^{[l]}, x_A^{[1]}, \dots, x_A^{[l-1]})$$

Apply both a random selection and dependence on previous solutions.

New information on the arcs of the construction graph guides the search. The ART uses memory, the GRASP randomization, the AS both.

5.2 Adaptive Research Technique ART

It was proposed by Patterson et al. (1998) for the CMSTP.

When **deceivingly good elements** are included in the **first steps** the **final solution can be quite bad**.

Aiming to avoid that

- The roll-out approach makes a look-ahead on each possible element (but a single step can be insufficient to identify the misleading ones).
- The **ART forbids some elements to drive subset x on the right path** in the search space (how to identify the misleading elements?).

Forbidding elements of the previous solution guarantees to obtain different solutions.

The **prohibitions are temporary**, with an expiration time of L iterations; otherwise, building feasible solutions would soon become impossible.

How does it work? Define a basic constructive heuristic A .

Let T_i be the **starting iteration of the prohibition** for each element $i \in B$ and x^* be the best solution found. Essentially a vector that keeps track of when the prohibition starts.

Set $T_i = -\infty$ for all $i \in B$ to indicate that **no element is forbidden**.

At each iteration $l \in \{1, \dots, \ell\}$

1. Apply heuristic A **forbidding all elements i such that $l \leq T_i + L$** (the current iteration must be less than the value at which the prohibition for that element started + the expiration time L , thus all prohibitions older than L iterations automatically expire); let $x^{[l]}$ be the **resulting solution**.
2. If $x^{[l]}$ is better than x^* , set $x^* := x^{[l]}$ and save $T_i - l$ (to know which elements in the solution were forbidden and for how much time) for all $i \in B$.
3. **Decide** which **elements to forbid** and set $T_i = l$ for them: each element is forbidden with probability π (any better ideas?).
4. Make minor tweaks to L, π or T_i .

At the end, **return** x^* (best solution found).

Example: ART for the SCP

$$c \quad \boxed{25 \ 6 \ 8 \ 24 \ 12}$$

$$A \quad \begin{array}{ccccc} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array}$$

Let $L = 2$, $\pi = 0.15$, pseudo-random numbers $0.1, 0.9, 0.4, 0.5, 0.1, 0.2, \dots$

1. the basic heuristic finds solution $x^{[1]} = \{2, 3, 5, 4\}$ of cost $f(x^{[1]}) = 50$; forbid column 2 (because $0.1 \leq \pi < 0.9, 0.4$ and 0.5)
2. the basic heuristic finds solution $x^{[2]} = \{3, 1\}$ (2 is forbidden) of cost $f(x^{[2]}) = 33$; forbid column 3 (because $0.1 \leq \pi < 0.2$)
3. the basic heuristic finds solution $x^{[3]} = \{1\}$ (3 and 2 are forbidden) of cost $f(x^{[3]}) = 25$, that is optimal
4. ...

An unlucky sequence could forbid column 1 at step 2.

5.2.1 Parameter tuning

The ART has **three basic parameters**

- the **total number of iterations** ℓ (tuned mainly by the available time)
- the **length L of the prohibition**
- the **probability π of the prohibition**

How to assign effective values to the parameters?

The **experimental comparison** of different values is necessary but complex

1. it requires **long experimental campaigns**, because the number of configurations grows combinatorially with
 - the number of parameters
 - the number of tested values for each parameter(the more sensitive the result, the more values must be tested)
2. it **risks overfitting**, that is labeling as absolutely good values which are good only on the benchmark instances considered

The excess of parameters is an undesirable aspect, and often reveals an insufficient study of the problem and of the algorithm.

5.2.2 Diversification and intensification

Diversification aims to obtain different solutions at every iteration. The ART achieves it by forbidding elements of the previous solutions.
An excessive diversification can hinder the discovery of the optimum.

Intensification aims to focus the search on the more promising subsets.

Diversification and intensification play complementary roles.

Their relative strength can be tuned through the parameters based on

- **problem data:** assign
 - a **smaller probability** π_i to be forbidden (forbidding a few elements)
 - a **shorter expiration time** L_i of the prohibition (forbidding for a short time)
- **to promising elements** (e.g., cheaper ones). This can lead to intensification on promising solutions.
- **memory:**
 - assign a **smaller π_i or L_i** (i is never forbidden when $\pi_i = 0$ or $L_i = 0$) **to promising elements** (e.g., appearing in the best known solutions)
 - **periodically restart the algorithm** with the $T_i - l$ values associated with the best known solution, instead of $T_i = -\infty$

This is trying to learn which parameters are “good” through past solutions.

5.3 Semi-greedy heuristics

A **nonexact constructive algorithm** has at least one step t which builds a subset $x^{(t)}$ **not included in any optimal solution** (since it's not the best, at one step it takes the wrong choice).

Since the element selected is the **best according to the selection criteria**

$$i^* = \arg \min_{i \in \Delta_A^+(x)} \varphi_A(i, x)$$

necessarily $\varphi_A(i, x)$ is **incorrect**, but probably **not completely wrong**.

The **semi-greedy algorithm** (Hart and Shogan, 1987) assumes that **elements that lead to the optimum are very good for $\varphi_A(i, x)$** , even if not strictly the best.

How to know which one?

If it is not possible to refine $\varphi_A(i, x)$

- Define a **suitable probability distribution** on $\Delta_A^+(x)$ favoring the elements with the **best values** of $\varphi_A(i, x)$.
- **Select $i^*(\omega)$ according to the distribution function.**

Since the set of alternative choices is finite, this means to **assign**

- **probability $\pi_A(i, x)$ to arc $(x, x \cup \{i\})$ of the construction graph** (with a sum equal to 1 for the outgoing arcs of each node, total probability)

$$\sum_{i \in \Delta_A^+(x)} \pi_A(i, x) = 1 \text{ for all } x \in \mathcal{F}_A : \Delta_A^+(x) \neq \emptyset$$

I have to get out of a node, each arc has a certain probability.

- **higher probabilities to the better elements** for the selection criteria

$$\varphi_A(i, x) \leq \varphi_A(j, x) \Leftrightarrow \pi_A(i, x) \geq \pi_A(j, x)$$

for each $i, j \in \Delta_A^+(x)$, $x \in \mathcal{F}_A$.

If the selection criteria is less, the probability must be more (because it means that it could be a better path).

This heuristic approach has important properties

- It can **reach an optimal solution** if there is a **path from \emptyset to X^*** (this is a basic condition).
- It can be **reapplied several times obtaining different solutions** and the probability to reach the optimum grows gradually (each iteration decreases the probability of missing an optimal path).

5.3.1 Convergence to the optimum

The probability of

- **following a path γ** is the **product of the probabilities on the arcs**

$$\prod_{y, y \cup \{i\} \in \gamma} \pi_A(i, y)$$

Each path is the product of every one of its arcs.

- **obtaining a solution x** is the **sum of those of the paths Γ_x reaching x**

$$\sum_{\gamma \in \Gamma_x} \prod_{y, y \cup \{i\} \in \gamma} \pi_A(i, y)$$

The probability of a solution is the sum of the probability of all its possible paths.

This implies that the **probability to reach the optimum**:

1. **Is nonzero if and only if there exists a path of nonzero probability from \emptyset to X^* .**
2. **Increases as $\ell \rightarrow +\infty$** (the probability of not reaching it decreases gradually)

It tends to 1 for probabilistically approximately complete algorithms.

In this context, a **random walk** is a constructive metaheuristic in which **all the arcs going out of the same node have equal probability**

- it finds a path to the optimum with probability 1 (if one exists)
- the time required can be extremely long. The exhaustive algorithm is exact and requires finite time

A **deterministic** constructive heuristic sets **all probabilities to zero** except for those on the **arcs of a single path**

- it finds the optimum only if it enjoys specific properties
- it finds the optimum in a single run

You just have the minimum selection criteria, which is chosen deterministically.

Randomized heuristics that **favor promising arcs and penalize the others**

- accelerate the average convergence time
- decrease the guarantee of convergence in the worst case

There is a **trade-off between expected and worst result**.

Arcs with **zero probability** can block the path to the optimum.

Arcs with **probability converging to zero** reduce the probability of finding it.

5.4 GRASP and Semi-greedy

GRASP, that is **Greedy randomized Adaptive Search Procedure** (Feo and Resende, 1989) is a more sophisticated variant of the semi-greedy heuristic

- **Greedy** indicates that it uses a constructive basic heuristic
- **Randomized** indicates that the basic heuristic makes random steps
- **Adaptive** indicates that the heuristic uses an adaptive selection criteria $\varphi_A(i, x)$, depending also on x (not strictly necessary)
- **Search** indicates that it alternates the constructive heuristic and an exchange heuristic (differently from the semi-greedy approach)

The use of auxiliary exchange heuristics allows strongly better results.

What probability function? Several functions $\pi_A(i, x)$ are monotonous with respect to $\varphi_A(i, x)$

$$\varphi_A(i, x) \leq \varphi_A(j, x) \Leftrightarrow \pi_A(i, x) \geq \pi_A(j, x)$$

Better value of the selection criteria correspond to larger probabilities.

- **uniform probability:** each arc going out of x has the same $\pi_A(i, x)$; the algorithm performs a random path in \mathcal{F}_A (random walk).

- **Heuristic-Biased Stochastic Sampling (HBSS):**

- sort the arcs going out of x by nonincreasing (best first) values of $\varphi_A(i, x)$
- assign a decreasing probability according to the position in the order based on a simple scheme (linear, exponential, ecc...)

Assign a probability to each arc depending on its position in the “ranking list”.

- **Restricted Candidate List (RCL):**

- sort the arcs going out of x by nonincreasing values of $\varphi_A(i, x)$
- insert the best arcs in a list (How many?)
- assign uniform probability to the arcs of the list, zero to the others

It just takes a subset containing the best arcs.

The most common strategy is the RCL, even if the zero probability arcs potentially cancel the global convergence to the optimum.

5.4.1 Definition of the RCL

Two main strategies are used to define the RCL

- **Cardinality:** the RCL includes the **best μ elements of $\Delta_A^+(x)$** , where $\mu \in \{1, \dots, |\Delta_A^+(x)|\}$ is a parameter fixed by the user
 - $\mu = 1$ yields the constructive basic heuristic (takes only the best)
 - $\mu = |\Delta_A^+(x)|$ (i. e., $|\Delta_A^+(x)|$ for each x) yields the random walk (takes all the choices)

You decide that you want to take μ elements (usually fixed), each node will have μ outgoing arcs.

- **Value:** the RCL includes **all the elements of $\Delta_A^+(x)$ whose value is between φ_{min} and $(1 - \mu)\varphi_{min} + \mu\varphi_{max}$** where

$$\varphi_{min}(x) = \min_{i \in \Delta_A^+(x)} \varphi_A(i, x) \quad \varphi_{max} = \max_{i \in \Delta_A^+(x)} \varphi_A(i, x)$$

and $\mu \in [0; 1]$ is a parameter fixed by the user

- $\mu = 0$ yields the constructive basic heuristic (takes only the minimum)
- $\mu = 1$ yields the random walk (takes everything from minimum to maximum)

This takes all the solution in a certain “range of quality”, determined by the parameter μ and the overall total quality range.

Example: GRASP for the SCP

$$c \quad \boxed{25 \ 6 \ 8 \ 24 \ 12}$$

$$A \quad \boxed{\begin{array}{ccccc} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array}}$$

Let $\mu = 2$ and the pseudo-random sequence be: 0.6, 0.8, ...

1. Start with the empty subset: $x^{(0)} = \emptyset$.
2. Build the RCL with columns 2($\varphi_2 = 2$) and 3($\varphi_3 = 4$); select column 3 (because $0.6 > 1/2$).
3. Build the RCL with columns 2($\varphi_2 = 3$) and 1($\varphi_1 = 6.25$); select column 1 (because $0.8 > 1/2$).
4. The solution obtained is $x = \{3, 1\}$ of cost $f(x) = 33$.

With $\mu = 2$, the optimal solution cannot be obtained; while with $\mu = 3$ it can be (the first column is the third in respect to the first selection criteria).

The optimum can be found with $\mu = 2$ if a destructive phase is applied.

5.4.2 Reactive parameter tuning

Once again there are **parameters to tune**:

- the **number of iterations** ℓ (for each “generation”)
- the **value** μ determining the **size of the RCL**

An idea to exploit memory is to **learn from the previous results** (reactive because it reacts to previous results)

1. Select m **configurations of parameters** μ_1, \dots, μ_m and set $\ell_r = \ell/m$ (divide the whole set of iteration in m parts, one for each parameter).
2. Run each configuration μ_r for ℓ_r iterations (same number for each, obtaining ℓ_r different solutions).
3. **Evaluate the mean** $\bar{f}(\mu_r)$ of the **results obtained with** μ_r (results obtained in the iterations with a certain parameter).
4. **Update the number of iterations** ℓ_r for each μ_r based on $\bar{f}(\mu_r)$

$$\ell_r = \frac{\frac{1}{\bar{f}(\mu_r)}}{\sum_{s=1}^m \frac{1}{\bar{f}(\mu_s)}} \ell \quad \text{for } r = 1, \dots, m$$

increasing it for the more effective configurations (more iterations to good configurations, less to bad ones).

The formula above makes a number $\in [0, 1]$ (then multiplied by the number of iterations ℓ) which is small for big values of $\bar{f}(\mu_r)$ and vice versa (minimization problem).

5. **Repeat the whole process**, going back to point 2, for R times (total number of “generation”).

Other schemes use scores based on the number of best known results.

You set m different parameters and you adjust the time dedicated to each based on their performance.

5.5 Cost perturbation methods

Instead of forbidding/forcing some choices, or modifying their probability, it is possible to **modify the appeal of the available choices** (selection criteria).

Given a basic constructive heuristic A , at each step of iteration l

- **tune the selection criteria** $\varphi_A(i, x)$ with a factor $\tau_A^{[l]}(i, x)$

$$\psi_A^{[l]}(i, x) = \frac{\varphi_A(i, x)}{\tau_A^{[l]}(i, x)}$$

- **update** $\tau_A^{[l]}(i, x)$ based on the previous solutions $x^{[1]}, \dots, x^{[l-1]}$

The selection criteria is multiplied by a factor τ associated to the arcs $((i, x)$ like the selection criteria) and can vary from iteration to iteration (${}^{[l]}$)

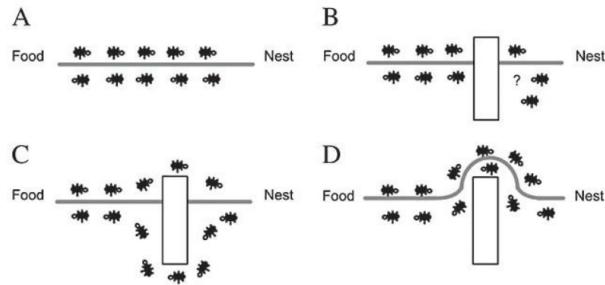
The elements with a better $\varphi_A(i, x)$ tend to be favored, but $\tau_A^{[l]}(i, x)$ tunes this effect, promoting

- **intensification** if $\tau_A^{[l]}(i, x)$ increases for the most frequent elements; this favors solutions similar to the previous ones (if the solutions are good)
- **diversification** if $\tau_A^{[l]}(i, x)$ decreases for the most frequent elements; this favors solutions different from the previous ones

5.6 Ant Colony Optimization

It was devised by Dorigo, Maniezzo and Colomi in 1991 drawing inspiration from the **social behavior of ants**.

Stigmergy = indirect communication among different agents who are influenced by the results of the actions of all agents.



They follow the better (stronger) trail.

Each agent is an application of the basic constructive heuristic

- it leaves a **trail on the data** depending on the solution generated
- it **performs choices influenced by the trails** left by the other agents

The choices of the agent have also a **random component**.

5.6.1 Trail

As in the semi-greedy heuristic

- a basic constructive heuristic A is given
- each step performs a partially random choice

Differently from the semi-greedy heuristic

- **Each iteration l runs h times heuristic A** (population).
- **All the choices of $\Delta_A^+(x)$ are feasible** (there is no RCL).
- The **probability $\pi_A(i, x)$ depends** on
 1. the **selection criteria** $\varphi_A(i, x)$
 2. **auxiliary information** $\tau_A(i, x)$ denoted as **trail** produced in previous iterations (sometimes by other agents in the same iteration)

The **trail** is **uniform at first** ($\tau_A(i, x) = \tau_0$), and **later tuned**

- **increasing it to favor promising choices**
- **decreasing it to avoid repetitive choices**

For the sake of simplicity, the trail $\tau_A(i, x)$ is not associated to each arc $(x, x \cup \{i\})$ (the associated data structure would be exponentially large and always way too big), but is the same for blocks of arcs (e.g., depending only on i , each element of the ground set).

5.6.2 Random choice

Instead of selecting the best element according to criteria $\varphi_A(i, x)$, **i is extracted from $\Delta_A^+(x)$ with probability**

$$\pi_A(i, x) = \frac{\tau_A(i, x)^{\mu_\tau} \eta_A(i, x)^{\mu_\eta}}{\sum_{j \in \Delta_A^+(x)} \tau_A(j, x)^{\mu_\tau} \eta_A(j, x)^{\mu_\eta}}$$

At each step the probability is given by the values of the trail τ and the values η related to the selection criteria.

where

- The **denominator normalizes the probability**.
- The **visibility** is the **auxiliary function**

$$\eta_A(i, x) = \begin{cases} \varphi_A(i, x) & \text{for maximization problems} \\ \frac{1}{\varphi_A(i, x)} & \text{for minimization problems} \end{cases}$$

The promising choices have larger visibility.

- The parameters μ_τ and μ_η **tune the weights** of the two terms

You are taking with larger probability elements that have a better selection criteria but that are also associated with a larger trail (they have been considered a lot in the previous iterations).

Balancing given and learned information: The original Ant System tunes the probabilities with parameters μ_η and μ_τ that control the amount of randomness

- $\mu_\eta \approx 0$ and $\mu_\tau \approx 0$ push towards randomness
- large values of μ_η and μ_τ push towards determinism
(favor $\arg \max_{i \in \Delta_A^+(x)} \tau_A(i, x)^{\mu_\tau} \eta_A(i, x)^{\mu_\eta}$)

and the relative weight of the data and of memory

- $\mu_\eta \gg \mu_\tau$ favors the data, simulating the basic constructive heuristic which makes sense when the known solutions are not very significant
- $\mu_\eta \ll \mu_\tau$ favors memory, keeping close to the previous solutions which makes sense when the known solutions are very significant

(assuming $\tau_A(i, x) > 1$ and $\eta_A(i, x) > 1$).

The **Ant Colony System** variant splits the selection into two phases

1. Decide the selection procedure

- with probability q , to choose i deterministically
- with probability $(1 - q)$, choose i stochastically

where parameter q tunes the randomness

- $q \approx 0$ favors random choices
- $q \approx 1$ favors deterministic choices

2. Apply the selection procedure

- the deterministic one selects the best element

$$i^* = \arg \max_{i \in \Delta_A^+(x)} \tau_A(i, x) \eta_A(i, x)^{\mu_n}$$

- the stochastic one select a random element with probabilities

$$\pi_A(i, x) = \frac{\tau_A(i, x) \eta_A(i, x)^{\mu_n}}{\sum_{j \in \Delta_A^+(x)} \tau_A(j, x) \eta_A(j, x)^{\mu_n}}$$

where parameter μ_n tunes the relative weight of data and memory

- $\mu_n \gg 1$ favors the data
- $\mu_n \ll 1$ favors memory

(setting $\mu_\tau = 1$ as a form of normalization).

5.6.3 Trail update

At each iteration ℓ

1. **run h instances** of the basic heuristic A
2. **select a subset $\tilde{X}^{[l]}$ of the solutions obtained**, in order to favor their elements in the following iterations
3. **update the trail** according to the formula

$$\tau_A(i, x) := (1 - \rho)\tau_A(i, x) + \rho \sum_{y \in \tilde{X}^{[l]}: i \in y} F_A(y)$$

The new trail derives from the current one with a convex combination of an additional term based off the elements of the solutions in the subset chosen and their relative fitness.

where

- $\rho \in [0; 1]$ is an **oblivion parameter** (number that reduces the current trail to give space to the new contribution)
- $F_A(y)$ is a **fitness function** expressing the **quality of solution** y (such that $F > \tau$: e.g., $F(y) = Q/f(y)$ for a suitable constant Q ; big if solution good, small if solution bad)

The new trail becomes a combination of the old one and a factor which scales with the quality of the solution in which a certain element is included.

The **purpose of the update** is to

1. **increase the trail on the elements of specific solutions** ($y \in \tilde{X}^{[l]}$)
2. **decrease the trail on the other elements**

The Oblivion parameter: $\rho \in [0; 1]$ tunes the behavior of the algorithm:

- **diversification:** a high oblivion ($\rho \approx 1$) cancels the current trail based on the intuition that
 - the solutions obtained are not trustworthy
 - different solutions should be explored
- **intensification:** a low oblivion ($\rho \approx 0$) preserves the current trail based on the intuition that
 - the solutions obtained are trustworthy
 - similar solutions should be explored

Selection of the influential solutions: $\tilde{X}^{[l]}$ collects the solutions around which the search will be intensified

- the classical Ant System considers all the solutions of iteration $l - 1$
- the **elitist methods** consider the best known solutions
 - the best solution of iteration $l - 1$
 - the best solution of all iterations $< l$

You look for an “elite” solution, the best solution in respect to something (all iterations/this iteration)

The elitist methods

- find **better results in shorter time**
- require **additional mechanisms to avoid premature convergence**

Some variants of the Ant System:

- **MAX – MIN Ant System:** imposes on the trail a limited range of values $[\tau_{min}; \tau_{max}]$, experimentally tuned.
- **HyperCube Ant Colony Optimization (HC-ACO):** normalizes the trail between 0 and 1.
- Ant Colony System: updates the trail on two levels
 - the **global update** (already seen) modifies it at each iteration ℓ (the purpose is to intensify the search)
 - the **local update** updates the trail at each application g of the basic heuristic in order to discourage identical choices in the following

$$\tau_A(i, x) := (1 - \rho)\tau_A(i, x) \text{ for each } i \in x^{[l, g]}$$

The purpose is to diversify the search.

5.6.4 Convergence to the optimum

Some variants of the Ant System **converge to the optimum** with probability 1 (Gutjahr, 2002).

The **analysis is based on the construction graph**

- The **trail** $\tau_A(i, x)$ is laid down **on the arcs** $(x, x \cup \{i\})$.
- **No information from the data** is used, that is $\eta_A(i, x) \equiv 1$ (only the trail, no data; this strange assumption simplifies the computation, but is not necessary).
- $\tau^{[l]}$ is the **trail function** at the **beginning of iteration** l .
- $\gamma^{[l]}$ is the **best path** on the graph at the **end of iteration** l .
- $(\tau^{[l]}, \gamma^{[l-1]})$ is the **state of a non-homogeneous Markov process**:
 - the **probability of each state** depends only on the **previous iteration**
 - the **process is non-homogeneous** because the **dependency varies with** l

The **proof concludes** that for $l \rightarrow +\infty$, with probability 1

1. the **best path found** γ **is one of the optimal paths** in \mathcal{F}
2. the **trail** τ **tends to a maximum along** γ , to zero on the other arcs

provided that a suitable parameter tuning is adopted.

First variant with global convergence: The trail is updated with a variable coefficient of oblivion

$$\tau^{[l]}(i, x) := \begin{cases} (1 - \rho)^{[l-1]} \tau^{[l-1]}(i, x) + \rho^{[l-1]} \frac{1}{|\gamma^{[l-1]}|} & \text{if } (x, x \cup \{i\}) \in \gamma^{[l-1]} \\ (1 - \rho)^{[l-1]} \tau^{[l-1]}(i, x) & \text{otherwise} \end{cases}$$

where $\gamma^{[l-1]}$ is the **best path found** in the graph **up to iteration $l-1$** and $|\gamma^{[l-1]}|$ is the **number of its arcs** (to normalize the trail).

The idea is that if you're considering the best known path up until that point $((x, x \cup \{i\}) \in \gamma^{[l-1]}, \text{ elitist strategy})$ then you are going to increase the trail on that path by a certain amount (1 divided by the number of arcs on that path). If an arc is not on the best known path you just decrease the trail.

Differences: the objective function is not considered since data is not taken into account, and the oblivion parameter is not fixed (decreases, but not very fast).

If the oblivion decreases slowly enough

$$\rho^{[l]} \leq 1 - \frac{\log l}{\log(l+1)} \quad \text{and} \quad \sum_{l=0}^{+\infty} \rho^{[l]} = +\infty$$

First half puts a bound to the value of the parameter, which will be forced to decrease. The second part means that the sum of all parameters should diverge, i.e. it should not decrease too fast (otherwise there's the risk of a premature convergence).

Then with probability 1 **the state converges to** (τ^*, γ^*) , where

- γ^* is an optimal path in the construction graph
- $\tau^*(i, x) = 1/|\gamma^*|$ for $(x, x \cup \{i\}) \in \gamma^*$, 0 otherwise

Second variant with global convergence: Alternatively, if the **oblivion** ρ remains constant, but the **trail is forced a slowly decreasing minimum threshold**

$$\tau(i, x) \geq \frac{c_l}{\log(l+1)} \text{ and } \lim_{l \rightarrow +\infty} c_l \in (0; 1)$$

then with probability 1 **the state converges to** (τ^*, γ^*) .

Here the oblivion is restricted by the minimum threshold.

The trail must always be larger than the first term and it means that the trail must not decrease to zero too fast, otherwise there could be a premature convergence, but its limit must be finite, so it must not be too slow.

In practice, all algorithms proposed so far in the literature

- associate the trail to groups of arcs $(x, x \cup \{i\})$ (e.g., to each element i)
- use constant values for parameters ρ and τ_{min}

therefore do not guarantee convergence.

The trail τ , and therefore π , can tend to zero on every optimal path.

6 Exchange Algorithms

In Combinatorial Optimization every solution x is a subset of B .

An exchange heuristic **updates a current subset** $x^{(t)}$ step by step

1. **Start** from a **feasible solution** $x^{(0)} \in X$ found somehow (often by a constructive heuristic).
2. Generate a **family of feasible solutions** by exchanging elements, i.e. add subsets A external to $x^{(t)}$ and delete subsets D internal to $x^{(t)}$

$$x'_{A,D} = x \cup A \setminus D \text{ with } A \subseteq B \text{ and } D \subseteq x$$

3. Use a **selection criteria** $\varphi(x, A, D)$ to choose the subsets to exchange

$$(A^*, D^*) = \arg \min_{(A, D)} \varphi(x, A, D)$$

4. **Perform** the chosen **exchange** to generate the new current solution

$$x^{(t+1)} := x^{(t)} \cup A^* \setminus D^*$$

5. If a **termination condition** holds, terminate; otherwise, go back to point 2

6.1 Neighborhoods

An exchange heuristic is defined by:

1. The **pairs of exchangeable subsets** (A, D) in every solution x , i.e. the solutions generated by a single exchange starting from x .
2. The **selection criteria** $\varphi(x, A, D)$.

Neighborhood $N : X \rightarrow 2^X$ is a function which **associates to each feasible solution** $x \in X$ a **subset of feasible solutions** $N(x) \subseteq X$.

The situation can be formally **described with a search graph** in which

- The **nodes** represent the **feasible solutions** $x \in X$.
- The **arcs connect each solution x to those of its neighborhood** $N(x)$, moving elements into and out of x (they are often denoted as moves).

Every run of the algorithm corresponds to a path in its search graph.

How does one define a neighborhood and select a move?

Essentially, it's the set of neighbor solutions, the one obtainable by one move, i.e. one set of exchanges.

6.1.1 Neighborhoods based on distance

Every solution $x \in X$ can be represented by its **incidence vector**

$$x_i = \begin{cases} 1 & \text{if } i \in x \\ 0 & \text{if } i \in B \setminus x \end{cases}$$

Associates 1 if the element is in the solution, 0 if it isn't; basically says which elements are in the solution and which ones aren't.

Hamming distance between two solutions x and x' is the **number of elements in which their incidence vectors differ**

$$d_H(x, x') = \sum_{i \in B} |x_i - x'_i|$$

It adds, for each element $i \in B$, 1 if the element is in only one of the solutions and 0 if the element is in both/none of the solutions (fancy math way to count the differences).

Referring to the subsets, $d_H(x, x') = |x \setminus x'| + |x' \setminus x|$; equivalent definition, this is the cardinality of the elements that belong to one solution but not to the other.

A typical definition of neighborhood, with an integer parameter k , is the **set of all solutions with a Hamming distance from x not larger than k**

$$N_{H_k}(x) = \{x' \in X : d_H(x, x') \leq k\}$$

Example: The KP instance with $B = \{1, 2, 3, 4\}$, $v = [5 4 3 2]$ and $V = 10$, has 13 feasible solutions out of 16 subsets

(0111)	(1111)	(0001)	(0000)
(0011)	(1011)	(1010)	(0110)
(1110)	(1001)	(1000)	(0101)
(1100)	(1101)	(0010)	(0100)

since subsets $\{1, 2, 3, 4\}$, $\{1, 2, 3\}$ and $\{1, 2, 4\}$ are unfeasible.

10 subsets (pink) have Hamming distance ≤ 2 from $x = \{1, 3, 4\}$ (blue).

The neighborhood $N_{H_2}(x)$ consists of the 7 feasible subsets in pink.
 $N_{H_2}(x)$ excludes

- the 3 crossed subsets in pink because they are unfeasible
- the 5 subsets in black because their Hamming distance from x is > 2

The neighborhood must include only feasible solutions which respect the Hamming distance.

6.1.2 Neighborhoods based on operations

Another common definition of neighborhood is obtained defining

- a **family \mathcal{O} of operations** on the solutions of the problem
- the **set of all solutions generated applying to x the operations of \mathcal{O}**

$$N_{\mathcal{O}}(x) = \{x' \in X : \exists o \in \mathcal{O} : o(x) = x'\}$$

Considering again the KP, \mathcal{O} can be defined as

- **adding** to x an element of $B \setminus x$
- **deleting** from x at most an element (just to impose $x \in N(x)$)
- **swapping** one element of x with one of $B \setminus x$

The resulting neighborhood $N_{\mathcal{O}}$ is related to those defined by the Hamming distance, but does not coincide with any of them

$$N_{H_1} \subset N_{\mathcal{O}} \subset N_{H_2}$$

The Hamming distance 1 is just one operation, with this definition we can do more stuff, so it's bigger.

As the distance-based ones, these **neighborhoods can be parameterized** considering **sequences of k operations** of \mathcal{O} instead of a single one

$$N_{\mathcal{O}_k}(x) = \{x' \in X : \exists o_1, \dots, o_k \in \mathcal{O} : o_k(o_{k-1}(\dots o_1(x))) = x'\}$$

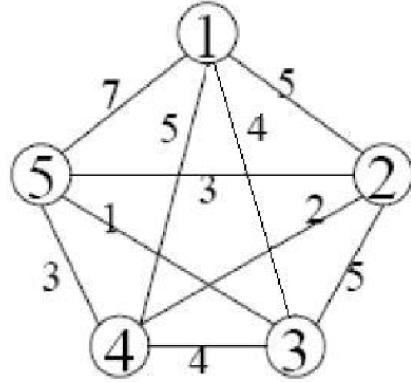
(bunch of operations, one after the other).

Essentially, define what you can do on a set (operations), and set a “number of operations away” distance metric.

6.1.3 Distance and operation-based

In general, an **operation-based** neighborhood includes solutions with **different Hamming distances from x** .

For the TSP one can define a neighborhood N_{S_1} including the solutions obtained swapping two nodes in their visit order



The neighborhood of solution $x = (3, 1, 4, 5, 2)$ is (5 vertices so $(5 \cdot 4)/2$ pairs):

$$N_{S_1}(x) = \{(\textcolor{red}{1}, 3, 4, 5, 2), (\textcolor{red}{4}, 1, \textcolor{red}{3}, 5, 2), (\textcolor{red}{5}, 1, 4, \textcolor{red}{3}, 2), (\textcolor{red}{2}, 1, 4, 5, \textcolor{red}{3}), (\textcolor{red}{3}, \textcolor{red}{4}, 1, 5, 2), (\textcolor{red}{3}, \textcolor{red}{5}, 4, \textcolor{red}{1}, 2), (\textcolor{red}{3}, \textcolor{red}{2}, 4, 5, \textcolor{red}{1}), (\textcolor{red}{3}, 1, \textcolor{red}{5}, 4, 2), (\textcolor{red}{3}, 1, \textcolor{red}{2}, 5, 4), (\textcolor{red}{3}, 1, 4, \textcolor{red}{2}, 5)\}$$

If the two nodes are adjacent, the modified arcs are $3 + 3$; otherwise, they are $4 + 4$.

This neighborhood cannot be defined as one with Hamming distance, in this case for this solution you include 4 arcs less and 4 more (to make the swap), 3 if the node were adjacent, so the Hamming distance between two of these solutions would be 8, 6 in the case of adjacent nodes.

We can see that this neighborhood doesn't coincide with N_{H_8} nor with N_{H_6} .

Relations: Sometimes the two definitions yield the same neighborhood

- for the MDP
 - N_{H_2} (solutions at Hamming distance equal to 2)
 - N_{S_1} (swap one element of x with one of $B \setminus x$)
- for the BPP
 - N_{H_2} (solutions at Hamming distance equal to 2)
 - N_{T_1} (transfer an object into a different container)
- for Max-SAT
 - N_{H_2} (solutions at Hamming distance equal to 2)
 - N_{F_1} (“flip” a variable: invert its truth assignment)

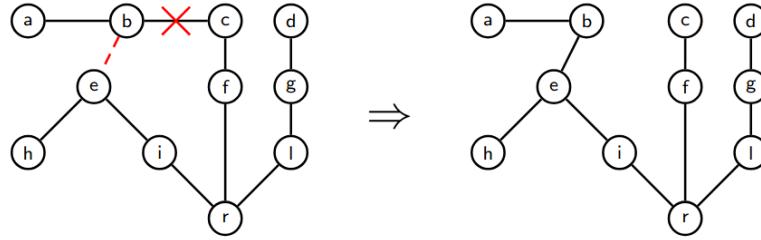
This is typical of problems with **solutions of fixed cardinality**:

- perform a **sequence of k swaps** between single elements ($|A| = |D| = 1$): k elements go into x and k elements go out
- the **Hamming distance** between the two extreme solutions **is** $\leq 2k$ (if all exchanged elements are different, it is exactly $2k$)

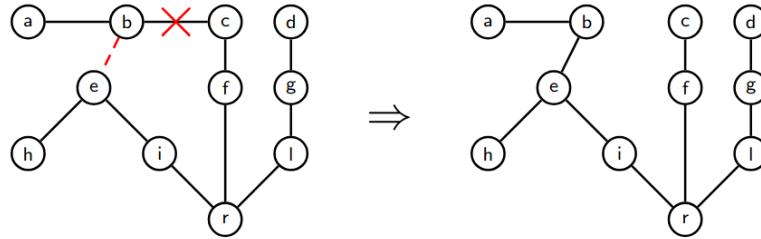
Different neighborhoods for the same problem: the CMST.

Different ground sets yield different neighborhoods. In the CMST it is possible to set $B = E$ or $B = V \times T$

- exchange **edges**: delete (b, c) , add (b, e)

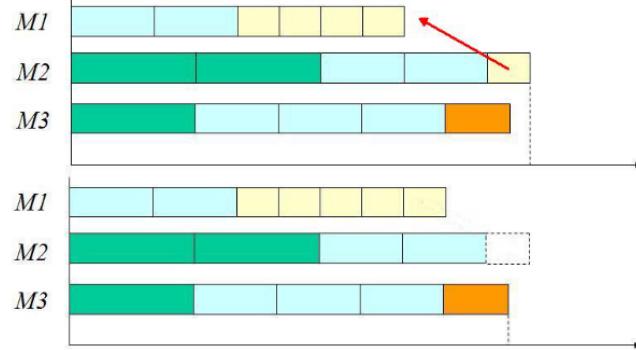


- exchange **vertices**: move e from subtree 2 to subtree 1, and recompute the two minimum spanning subtrees

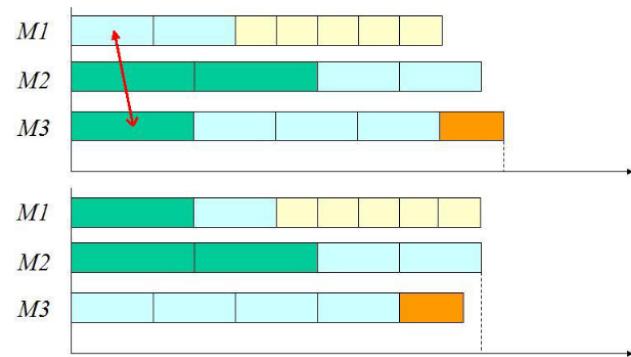


PSMP: For the PMSP it is possible to define

- the transfer neighborhood $N_{\mathcal{T}_1}$, based on the set \mathcal{T}_1 of **all transfers** of a task on another machine



- the swap neighborhood $N_{\mathcal{S}_1}$, based on the set \mathcal{S}_1 of the **swaps of two tasks** between two machines (one task for each machine)



6.1.4 Connectivity of the search graph

An exchange heuristic can return the optimum only if **every feasible solution** can **reach** at least **one optimal solution**, that is there is a path from x to X^* for every $x \in X$.

Such a search graph is denoted as **weakly connected** to the optimum.

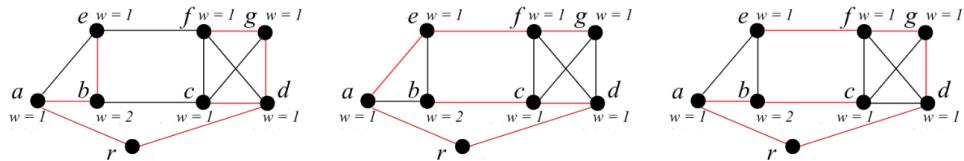
Since X^* is unknown, a **stronger condition** is often used: a search graph is **strongly connected** when it admits a path from x to y for every $x, y \in X$.

A **good neighborhood** should guarantee some **connectivity conditions**

- in the MDP, neighborhood N_{S_1} connects any pair of solutions with at most k swaps
- in the KP and the SCP, no neighborhood N_{S_k} gives that guarantee (feasible solutions can have any cardinality)
- the search graph becomes connected also in the KP and the SCP if swaps are combined with both additions and deletions

Strong connectivity is when every feasible solution is reachable from every other feasible solution. Everything is connected.

If feasibility is defined in a sophisticated way, exchanging, adding and deleting **single** elements can be **insufficient** to reach all solutions: the **unfeasible subsets** can **break** all paths between some feasible solutions.



If $V = 4$, only three solutions are feasible, all with two subtrees:

$$\begin{aligned} x &= \{(r, a), (a, b), (b, e), (r, d), (c, d), (d, g), (f, g)\} \\ x' &= \{(r, a), (a, e), (e, f), (r, d), (c, d), (b, c), (f, g)\} \\ x'' &= \{(r, a), (a, b), (e, f), (r, d), (b, c), (d, g), (f, g)\} \end{aligned}$$

The three solutions are mutually reachable only exchanging at least two edges at a time; exchanging only one yields unfeasible subsets.

6.2 Steepest descent (hill-climbing) heuristics

Abandoning the concept of neighborhood, let's consider how to move within them, what is the selection criteria?

The **simplest selection criteria** $\varphi(x, A, D)$ is the **objective function** (it is used in nearly all exchange heuristics).

When $\varphi(x, A, D) = f(x \cup A \setminus D)$, the **heuristic moves from $x^{(t)}$ to the best solution in $N(x^{(t)})$** (this is considering only the update in value, not recalculating the whole objective function, if it makes the computation easier).

To avoid cyclic behavior, **only strictly improving solutions** are accepted. You can't "go back", otherwise you risk visiting multiple times the same solutions (if an earlier solution was better why not going back to it?). Consequently, **the best known solution is the last visited one**.

Steepest Descent for minimization, Ascent/Hill-climbing for maximization. Note that the best solution is not saved, since it's always the last one.

Algorithm 6 Algorithm *SteepestDescent*($I, x^{(0)}$)

```

 $x := x^{(0)}$ 
Stop:= false
while Stop= false do
     $\bar{x} := \arg \min_{x' \in N(x)} f(x')$ 
    if  $f(\bar{x}) \geq f(x)$  then
        Stop:= true
    else
         $x := \bar{x}$ 
    end if
end while
return ( $x, f(x)$ )

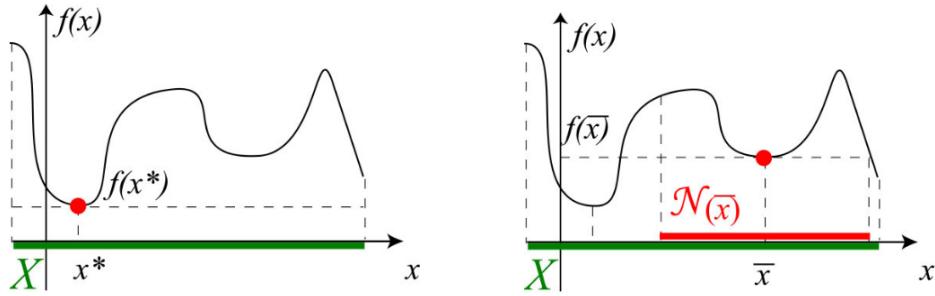
```

6.2.1 Local and global optimality

A steepest descent heuristic **terminates**, by definition, when it finds a **locally optimal solution**, that is a solution $\bar{x} \in X$ such that

$$f(\bar{x}) \leq f(x) \text{ for each } x \in N(\bar{x})$$

A local optimum is the best (technically not worse) solution inside of a neighborhood.



A globally optimal solution is always also locally optimal, but the opposite is not true in general: $X^* \subseteq \overline{X}_N \subseteq X$.

There could be a better solution outside of the neighborhood, so the set of locally optimal solution is dependent on the definition of neighborhood.

6.2.2 Exact neighborhood

Exact neighborhood is a neighborhood function $N : X \rightarrow 2^X$ such that **each local optimum is also a global optimum**

$$\overline{X}_N = X^*$$

Trivial case: the neighborhood of each solution coincides with the whole feasible region ($N(x) = X$ for each $x \in X$, if I have the whole set I'm going to get the optimum). It's a useless neighborhood: too wide to explore.

The exact neighborhoods are **extremely rare**; examples:

- exchange between edges for the Minimum Spanning Tree problem
- exchange between basic and nonbasic variables used by the simplex algorithm for Linear Programming

In general, the steepest descent heuristic does not find a global optimum.

Its effectiveness depends on the properties of search graph and objective.

Essentially is a neighborhood that includes the global optimum; in general it's hard to obtain.

6.2.3 Properties of the search graph

The properties of the graph can determine if our algorithm will find a “good” local optimum or a “bad” local optimum.

Some relevant properties for the effectiveness of an algorithm are

- The **size of the search space** $|X|$ (big space needs big time, small space needs small time).
- The **connectivity of the search graph** (as discussed above, maybe the optimal solution is in a not connected part of the graph).
- The **diameter of the search graph**, that is the number of arcs of the minimum path between the two farthest solutions: larger neighborhoods produce graphs of smaller diameter (but other factors exist: see the “smallworld” effect).

Consider neighborhood N_{S_1} for the symmetric TSP on complete graphs

- the search space includes $|X| = (n - 1)!$ solutions
- N_{S_1} (swap of two nodes) includes $\binom{n}{2} = n(n - 1)/2$ solutions
- the search graph is strongly connected and has diameter $\leq n - 2$: every solution turns into another after at most $n - 2$ swaps.

For example, $x = (1, 5, 4, 2, 3)$ becomes $x' = (1, 2, 3, 4, 5)$ in 3 steps

$$x = (1, 5, 4, 2, 3) \rightarrow (1, 2, 4, 5, 3) \rightarrow (1, 2, 3, 5, 4) \rightarrow (1, 2, 3, 4, 5) = x'$$

(the first node is always 1, the last one is automatically in place)

Other relevant properties:

- The **density of global optima** ($|X^*|/|X|$) and **local optima** ($|\bar{X}_N|/|X|$): if the local optima are numerous, it is hard to find the global ones.
How many global and local optima you have versus the total number of solutions.
- The **distribution of the quality $\delta(\bar{x})$ of local optima** (SQD diagram): if local optima are good, it is less important to find a global one.
- The **distribution of the locally optimal solutions in the search space**: if local optima are close to each other, it is not necessary to explore the whole space.

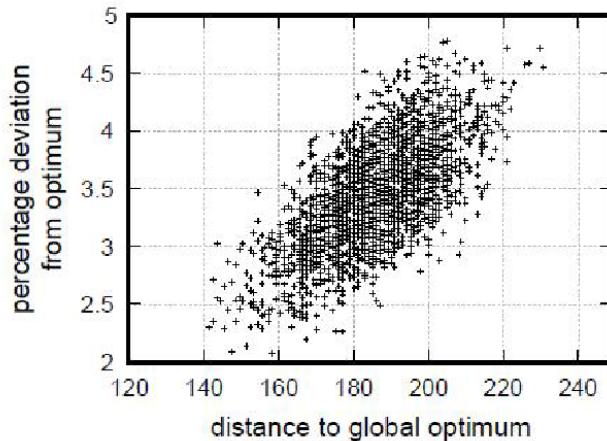
These indices would require an exhaustive exploration of the search graph.

In practice, one performs a sampling and these analyses

- require very long times
- can be misleading, especially if the global optima are unknown

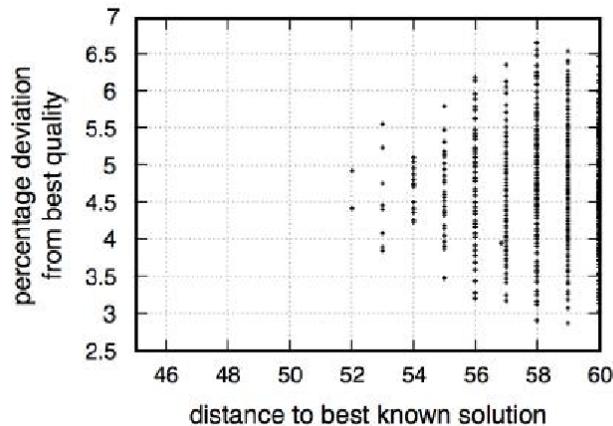
Example: the TSP. For the TSP on a complete symmetric graph with Euclidean costs

- The Hamming distance between two local optima is on average $\ll n$: the local optima concentrate in a small region of X .
- The Hamming distance between local optima on average exceeds that between local and global optima: the global optima tend to concentrate in the middle of local optima.
- The *FDC* diagram (Fitness-Distance Correlation) reports the quality $\delta(\bar{x})$ versus the distance from global optima $d_H(\bar{x}, X^*)$ (for each local optima, on the horizontal axis is shown how far they are from the global, on the vertical axis there's the percentage deviation δ): if they are correlated, better local optima are closer to the global ones.



This diagram teaches that if, after you found a local optimum, it's good to (somehow) continue and intensify the search because you will probably move towards the global optimum.

For the Quadratic Assignment Problem (QAP), the situation is different



If quality and closeness to the global optima are **strongly correlated**

- It is **profitable to build good starting solutions**, because they drive the search near a good local optimum.
- It is **better to intensify** than to diversify.

If the **correlation is weak**

- A good **initialization is less important**.
- It is **better to diversify** than to intensify

6.3 Landscape

The **landscape** is the **triplet** (X, N, f) , where

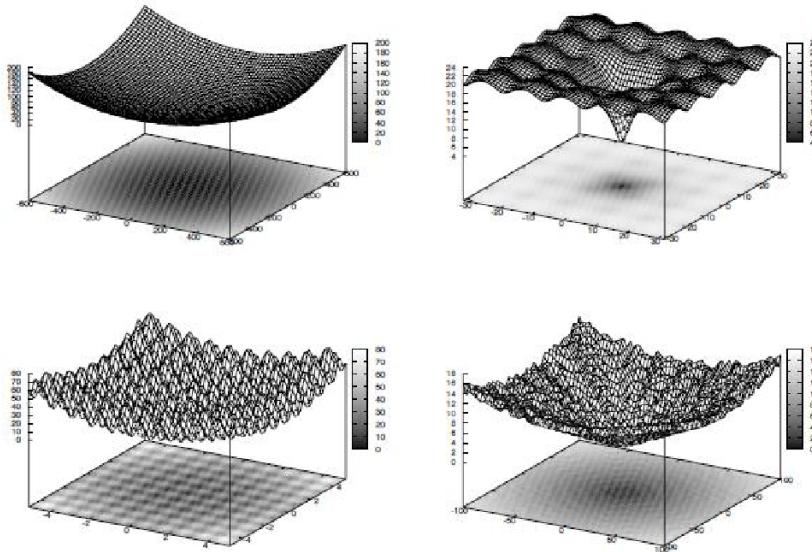
- X is the **search space**, or the set of feasible solutions
- $N : X \rightarrow 2^X$ is the **neighborhood function**
- $f : X \rightarrow \mathbb{N}$ is the **objective function**

It is the **search graph with node weights given by the objective**. The first two terms correspond to the search graph, the last one gives weights to the nodes.

The **effectiveness** of steepest descent **depends** on the **landscape**

- **smooth landscapes** yield few local optima, possibly of good quality, hence to **good results**
- **rugged landscapes** yield several local optima of widespread quality, hence to **bad results**

There is a **great variety of landscapes**, very different from one another



6.3.1 Autocorrelation coefficient

The **complexity of a landscape** can be **empirically estimated**

1. performing a **random walk** in the **search graph**
2. determining the **sequence of values** of the **objective** $f^{(1)}, \dots, f^{(t_{max})}$
3. computing the **sample mean**

$$\bar{f} = \frac{\sum_{t=1}^{t_{max}} f^{(t)}}{t_{max}}$$

4. computing the **empirical autocorrelation coefficient**

$$r(i) = \frac{\sum_{t=1}^{t_{max}-i} (f^{(t)} - \bar{f})(f^{(t+i)} - \bar{f})}{\frac{\sum_{t=1}^{t_{max}} (f^{(t)} - \bar{f})^2}{t_{max}}}$$

This is trying to relate the values of the objective function to the mean, with the goal of understanding how much these values vary; the denominator is the mean square error; the numerator is trying to know the difference between a step and the next, that way it can describe if the landscape is a smooth curve or a really rugged one (both could differ the same from the mean)

That **relates the difference of the objective values in the solutions visited with the distance between these solutions along the walk.**

This is trying to give a function to describe the “ruggedness” of a landscape.

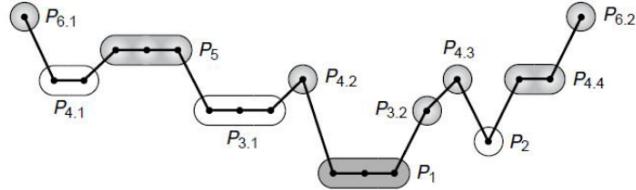
Idea is:

- With $i = 0 \rightarrow r(0) = 1$ (perfect correlation at 0 distance); numerator and denominator are the same, increasing the i is considering “how much is the value changing?”.
- In general $r(i)$ **decreases** as the **distance i increases**.
- **If $r(i) \approx 1$ in a large range of distances, the landscape is smooth:**
 - the neighbor solutions have values close to the current one
 - there are few local optima
 - the steepest descent heuristic is effective
- **If $r(i)$ varies steeply, the landscape is rugged:**
 - the neighbor solutions have values far from the current one
 - there are many local optima
 - the steepest descent heuristic is ineffective

6.3.2 Plateau

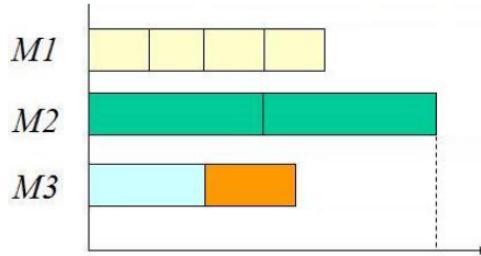
The search graph can be **partitioned** according to the **objective value**

- **plateau of value f** is each subset of solutions of value f that are adjacent in the search graph (a bunch of equal values together)



Large plateaus complicate the choice of the solution: most neighbors are equivalent, and the choice ends up depending on the visit order. An extremely uniform landscape is not an advantage.

Example: all transfers and swaps between machines 1 and 3 leave the objective value unchanged (most other moves worsen it)

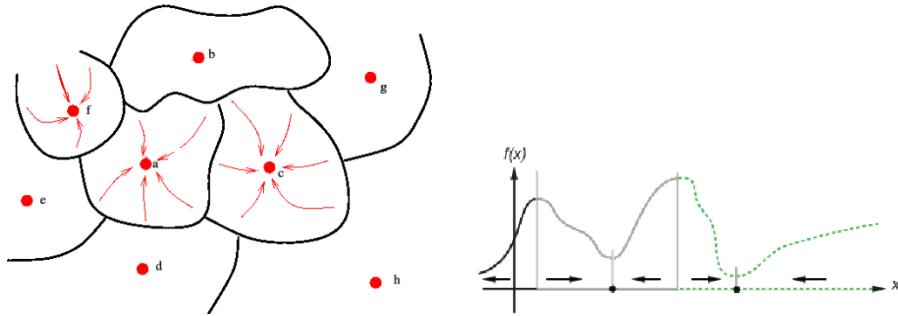


A plateau gives a lot of choices that are all effectively the same, making the objective function a bad criteria.

6.3.3 Attraction basins

Alternatively, the search graph can be **partitioned** into:

- **attraction basins** of the **locally optimal solutions** \bar{x} , that are the subsets of solutions $x^{(0)} \in X$ starting from which the steepest descent heuristic terminates in \bar{x} (if I start with a steepest descent in the basin, I'm going to get x since it's the closest local optimum)



The basins are separated by frontiers, on one side I get a local optimum, on the other side I get a different local optimum.

The steepest descent heuristic is

- **Effective** if the attraction **basins** are **few and large** (especially if the global optima have larger basins), even if the algorithm will have to take a lot of steps.
- **Ineffective** if the attraction **basins** are **many and small** (especially if the global optima have smaller basins).

6.4 Complexity

The complexity of the steepest descent heuristic depends on

- The **number of iterations** t_{max} **from $x^{(0)}$ to the local optimum found**, which depends on the structure of the search graph (width of the attraction basins) and is hard to estimate a priori.
- The **search for the best solution in the neighborhood** (\bar{x}), which depends on how the search itself is performed, but whose complexity estimation is usually standard.

6.4.1 The exploration of the neighborhood

It's the function in which you minimize the value of the objective.

Two strategies to explore the neighborhood are possible

1. **Exhaustive search:** evaluate **all the neighbor solutions**; the complexity of a single step is the product of
 - the number of neighbor solutions ($|N(x)|$)
 - the evaluation of the cost of each solution ($\gamma_f(|B|, x)$)

If it is not possible to generate only feasible solution:

- visit a superset of the neighborhood ($\tilde{N}(x) \supset N(x)$)
- for each element x , evaluate the feasibility ($\gamma_X(|B|, x)$)
- for the feasible ones, evaluate the cost ($\gamma_f(|B|, x)$)

2. **Efficient exploration** of the neighborhood **without a complete visit**: find the **best neighbor solution solving an auxiliary problem**. Only some special neighborhoods allow that.

Exhaustive visit of the neighborhood

Algorithm 7 Algorithm *SteepestDescent*($I, x^{(0)}$)

```

 $x := x^{(0)}$ 
Stop := false
while Stop = false do
     $\tilde{x} := x$  //  $\tilde{x} := \arg \min_{x' \in N(x)} f(x')$ 
    for each  $x' \in \tilde{N}$  do
        if  $x' \in N(x)$  then
            if  $f(x') < f(\tilde{x})$  then
                 $\tilde{x} := x'$ 
            end if
        end if
    end for
    if  $f(x') \geq f(x)$  then
        Stop := true
    else
         $x := \tilde{x}$ 
    end if
end while
return  $(x, f(x))$ 
```

The **complexity** of the neighborhood exploration **combines** three terms

1. $|\tilde{N}(x)|$: the **number of subsets visited**.
2. γ_X : the **time to evaluate their feasibility**.
3. γ_f : the **time to evaluate the objective** for a feasible solution.

6.4.2 Evaluating or updating the objective

The additive case: The first way to accelerate an exchange algorithm is to **minimize the time to evaluate the objective**: in particular, it is **faster to update $f(x)$ rather than to recompute it**.

The **update of an additive objective** $f(x) = \sum_{j \in x} \phi_j$ requires to

- Sum ϕ_i for each element $i \in A$, added to x .
- Subtract ϕ_j for each element $j \in D$, deleted from x

$$\delta f(x, A, D) = f(x \cup A \setminus D) - f(x) = \sum_{i \in A} \phi_i - \sum_{j \in D} \phi_j$$

Examples: swap of objects (KP), columns (SCP), edges (CMSTP), ...

This **update** has **two fundamental properties**:

- It takes **constant time for a constant number of elements** $|A| + |D|$.
- $\delta f(x, A, D)$ **does not depend on x** (only on the elements added and deleted, more about it later).

If the objective function is additive just add and subtract the changes.

The quadratic case: The MDP has a quadratic objective function: computing it costs $\Theta(n^2)$.

Moving from x to $x' = x \setminus \{i\} \cup \{j\}$ (neighborhood N_{S_1} , just making a swap), the **update is**

$$\delta f(x, i, j) = f(x \setminus \{i\} \cup \{j\}) - f(x) = \sum_{h, k \in x \setminus \{i\} \cup \{j\}} d_{hk} - \sum_{h, k \in x} d_{hk}$$

Which **depends on** $O(n)$ distance terms, related to points i and j (everything else cancels out).

There is a general trick **for the symmetric quadratic functions** with $d_{ii} = 0$

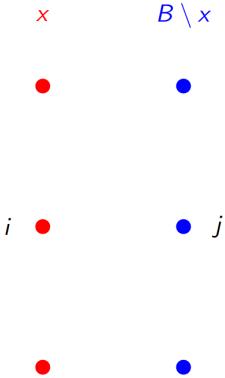
$$\begin{aligned} \delta f(x, i, j) &= \sum_{h \in x \setminus \{i\} \cup \{j\}} \sum_{k \in x \setminus \{i\} \cup \{j\}} d_{hk} - \sum_{h \in x} \sum_{k \in x} d_{hk} \implies \\ \implies \delta f(x, i, j) &= 2 \sum_{k \in x} d_{jk} - 2 \sum_{k \in x} d_{ik} - 2d_{ij} = 2(D_j(x) - D_i(x) - d_{ij}) \end{aligned}$$

If $D_\ell(x) = \sum_{k \in x} d_{\ell k}$ is known for each $\ell \in B$, **the computation takes** $O(1)$.

If you save the distance from every element to the solution (distance from every other element in the solution set) there's no need to calculate $D_j(x)$ and $D_i(x)$ (distance from point j and i to the solution) every time, leaving out only d_{ij} , which has to be subtracted (since we don't have i anymore) and can be done in constant time.

From the objective function we just need to add the distances relative to j (represented in $D_j(x)$), subtract the distances from i to the other elements in the solution ($D_i(x)$), and also subtract the distance from j to i , since i is not in the solution anymore. If we keep an auxiliary data structure for the distances $D_\ell(x)$ for every $\ell \in x$ then this update takes constant time. The auxiliary structures will have to be updated in $O(n)$ time each iteration.

Example: the MDP



Let us consider $f(x)/2$.

Evaluate the exchange

$$x \rightarrow x' = x \setminus \{i\} \cup \{j\}$$

with $i \in x$ and $j \in B \setminus x$ (swapping i from the solution with j from outside, i becomes blue, j becomes red).

$$f(x') = f(x) - D_i + D_j - d_{ij}$$

- the pairs including i are lost
- the pairs including j are acquired
- but the pair (i, j) is in excess

The cost is computed in $O(1)$ time for each solution.

Update of the data structures:

$$D_\ell = D_\ell - d_{\ll i} + d_{\ell j} \quad \ell \in B$$

For each element $\ell \in B$

- $d_{\ll i}$ disappears
- $d_{\ell j}$ appears

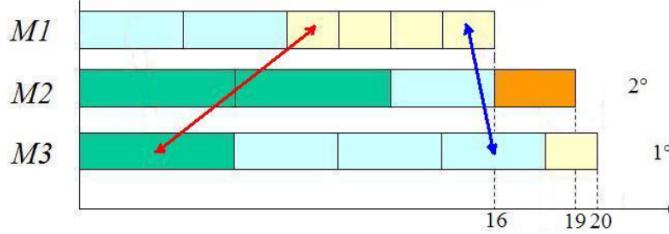
The auxiliary data structure is updated in $O(n)$ time for each iteration.

Nonlinear examples: Many nonlinear functions can be updated with similar tricks

- save aggregated information on the current solution $x^{(t)}$
- use it to compute $f(x')$ efficiently for each $x' \in N(x^{(t)})$
- update it when moving to the following solution $x^{(t+1)}$

Using the transfer ($N_{\mathcal{T}_1}$) and swap ($N_{\mathcal{S}_1}$) neighborhoods for the PMSP, the objective can be updated in constant time by managing

1. the completion time for each machine
2. the indices of the machines with the first and second maximum time



Consider the swap $o = (i, j)$ of tasks i and j (i on machine M_i , j on machine M_j)

- compute in constant time the new completion times: one increases, the other decreases (or both remain constant)
- test in constant time whether either exceeds the maximum
- if the maximum time decreases, test in constant time whether the other time or the second maximum time becomes the maximum

Once the neighborhood is visited and the exchange selected, update

- the two modified completion times (each one in constant time)
- their positions in a max-heap (each one in time $O(\log |M|)$)

Basically, compute the new the times, check if there's a new maximum, check if the maximum is still the maximum (if needed), then update the data structures.

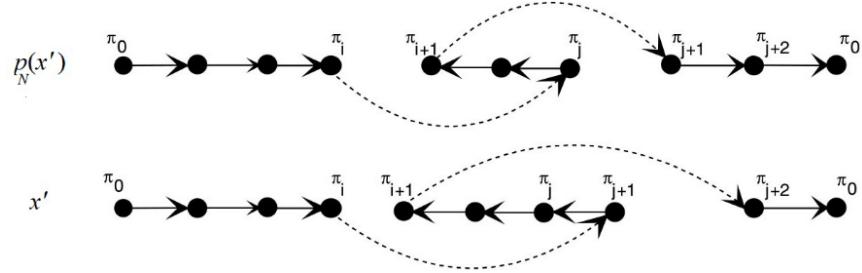
To do this we need to remember: completion time for each machine and which machines have the maximum and the second maximum time.

Use of local auxiliary information: The auxiliary information used to compute $f(x')$ can be

- **global**, that is referring to the **current solution x**
- **local**, that is referring to the **solution $p_N(x')$ visited before x'** in neighborhood $N(x)$ according to a suitable order

Consider the neighborhood $N_{\mathcal{R}_2}$ for the asymmetric TSP:

- the neighbor solutions differ from x for $O(n)$ arcs
- general neighbor solutions differ from each other for $O(n)$ arcs
- if the pairs of arcs (s_i, s_{i+1}) and (s_j, s_{j+1}) follow the lexicographic order, the reverted path changes only by one arc



The **variation** of $f(x)$ between **two generic neighbor** solutions is

$$\delta f(x, i, j) = c_{s_i, s_j} + c_{s_{i+1}, s_{j+1}} - c_{s_i, s_{i+1}} - c_{s_j, s_{j+1}} + c_{s_j \dots s_{i+1}} - c_{s_{i+1} \dots s_j}$$

but moving from exchange (s_i, s_j) to exchange (s_i, s_{j+1})

- the first four terms change, but they can be checked in constant time
- the last two terms can be updated in constant time

$$\begin{cases} c_{s_{j'} \dots s_{i+1}} &= c_{s_j \dots s_{i+1}} + c_{s_{j+1}, s_j} \\ c_{s_{i+1} \dots s_{j'}} &= c_{s_{i+1} \dots s_j} + c_{s_j, s_{j+1}} \end{cases}$$

Add the two arcs added (from i to j and from $i+1$ and $j+1$), remove the arcs that have been removed (from i to $i+1$ and from j to $j+1$), add the cost of the now reversed path, minus the cost of the original path. The last two terms (which can be big) can be updated in constant time by adding the new terms.

In constant time, with 6 terms the function can be updated.

Is it acceptable to explore the neighborhood in a predefined order?

6.5 Feasibility of the neighborhood

Defining neighborhoods with the Hamming distance or with operations can generate also **unfeasible subsets**, that **must be removed**

$$\tilde{N}_{H_k}(x) = \{x' \subseteq B : d(x', x) \leq k\} \supseteq N_{H_k}(x) = \tilde{N}_{H_k}(x) \cap X$$

$$\tilde{N}_{\mathcal{O}}(x) = \{x' \subseteq B : \exists o \in \mathcal{O} : o(x) = x'\} \supseteq N_{\mathcal{O}}(x) = \tilde{N}_{\mathcal{O}}(x) \cap X$$

(Examples: KP, BPP, SCP, CMSTP ...).

If it is not possible to avoid a priori the unfeasible subsets, one must

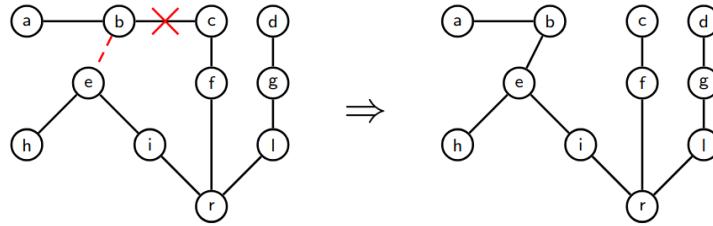
- **Test the feasibility of each element** of $\tilde{N}(x)$ to obtain $N(x)$.
- For the feasible elements, **evaluate** the **cost**.

The feasibility test can be made efficient with techniques similar to the ones used for the objective evaluation.

Example: update in constant time the total volume of a subset in the KP.

Example: The CMSTP. Consider the swap neighborhood N_{S_1} (add one edge, delete another)

- If the two edges are in the same branch, the solution remains feasible.
- If they are in different branches, one loses weight, the other acquires it: the variation is equal to the weight of the subtree transferred.



If each vertex saves the weight of its appended subtree, to test feasibility compare this weight with the residual capacity of the receiving branch (the weight appended to b with the residual capacity of the left branch).

Once the best exchange is performed, the information must be updated in time $O(n)$ visiting the old ancestors from c and the new ones from e .

If you know the total weight of the branch that has been cut (how many vertices are still appended) and confront that value with the residual capacity of the receiving branch.

You can save in a vector the total weight of the subtree appended to each vertex in the current solution. After a move you need to update all auxiliary information.

6.6 Partial saving of the neighborhood

When performing an **operation** $o \in \mathcal{O}$ on a **solution** $x \in X$ sometimes

- the **feasibility of the resulting solution** $o(x)$
- the **variation of the objective** $\delta f_o(x) = f(o(x)) - f(x)$

depend only on a part of x (possibly, very small).

For example, consider the swap neighborhood $N_{\mathcal{S}_1}$ for the CMST:

- add an edge $k \in B \setminus x$
- delete an edge $h \in x$

Two branches are involved: one acquires a subtree, the other loses it.

The **feasibility of swap** (i, j) depends on the **branches including i and j** : it is the **same in x and x'** and is **not affected by swap** (h, k)

$$\delta f_{i,j}(x) = \delta f_{i,j}(x')$$

For each operation $o \in \tilde{\mathcal{O}} \subset \mathcal{O}$ **and for each** $x' = o(x)$

- $o(x')$ is **feasible** if and only if $o(x)$ is **feasible**
- $\delta f_o(x') = \delta f_o(x)$

It is then **advantageous** to

1. compute and **save** $\delta f_o(x)$ **for every** $o \in \mathcal{O}$, that is keep the set of feasible exchanges and their associated values δf
2. **perform** the **best operation** o^* , and generate a new solution x'
3. recompute and **save** $\delta f_o(x')$ **only for** $o \in \mathcal{O} \setminus \tilde{\mathcal{O}}$, that is remove the exchanges on modified branches, recompute their values, and retrieve $\delta f_o(x')$ for all $o \in \tilde{\mathcal{O}}$ (their values are still correct)
4. go back to point 2

If the branches are numerous, $|\mathcal{O} \setminus \tilde{\mathcal{O}}| \ll |\mathcal{O}|$ and the saving is very strong.
It is typical of problems whose solution is a partition.

TLDR: save the variance for small operations, it will be the same across solutions which differ from one another in parts not affected by such operation.

6.7 Trade-off between efficiency and effectiveness

The **complexity** of an exchange heuristic **depends on three factors**

1. Number of iterations.
2. Cardinality of the visited neighborhood.
3. Computation of the feasibility and cost for the single neighbor.

The first two factors are clearly conflicting:

- A **small neighborhood** is **fast to explore**, but requires **several steps** to reach a local **optimum**.
- A **large neighborhood** requires **few steps**, but is **slow to explore**.

The optimal **trade-off is somewhere in the middle**: a neighborhood

- **large enough** to include **good solutions**
- **small enough** to be **explored quickly**

but it is hard to identify, because

- **efficiency quickly worsens** as size increases
- the resulting **solution also changes with the neighborhood** (large ones have better local optima)

It is also possible to **define a neighborhood** N and **tune its size**, you can modify it

- **Explore only a promising sub-neighborhood** $N' \subset N$. For example, if the objective function is additive, one can
 - add only elements $j \in B \setminus x$ of low cost ϕ_j
 - delete only elements $i \in x$ of high cost ϕ_i
- **Terminate the visit after finding a promising solution.** For example, the first-best strategy stops the exploration at the first solution better than the current one
If $f(\tilde{x}) < f(x)$ then $x := \tilde{x}$; Stop := *true*;

The **effectiveness depends on the objective**

- if the **cost of some elements influences very much the objective**, it is worth taking it into account, fixing or forbidding them

and on the **structure of the neighborhood**

- if the **landscape is smooth**, the first improving solution approximates well the best solution of the neighborhood: it is better to stop
- if the **landscape is rugged**, the best solution of the neighborhood could be much better: it is better to go on

6.8 Very Large Scale Neighborhood Search

Larger neighborhoods yield in general larger attraction basins, so that

- the **steepest descent** heuristic becomes very **effective**
- but the **exploration time is longer**

The **Very Large Scale Neighborhood (VLSN)** Search approaches have

- **neighborhoods exponential** in $|B|$ (or high-order polynomial)
- **explored** in low-order **polynomial time** (it's just another CO problem, finding the best solution in a finite set)

Two strategies allow limiting the computational time

1. select a neighborhood in which the **objective can be optimized** without an exhaustive exploration
2. **explore the neighborhood heuristically** and return a **promising neighbor solution**, instead of the best one

6.8.1 Efficient visit of exponential neighborhoods

Neighborhoods can be easily **parameterized**

$$N_{\mathcal{O}_k}(x) = \{x' \in X : x' = o_k(o_{k-1}(\dots o_1(x))) \text{ with } o_1, \dots, o_k \in \mathcal{O}\}$$

and it would be nice to **tune the number of operations k**

- increasing k when necessary to improve the current solution x
- decreasing k when sufficient to improve the current solution x

The idea is to **define a composite move** as a **set of elementary moves** (that is a combinatorial optimization problem).

Finding the optimal solution in such neighborhoods **requires solving an auxiliary problem**, typically on a matrix or graph

- **set packing:** Dynasearch
- **negative cost circuit:** cyclic exchanges
- **shortest path:** ejection chains, order-and-split

Such auxiliary tools are usually defined improvement matrices or graphs.

Combining elementary moves into composite ones: An operation $o \in \mathcal{O}$ usually modifies only some components of solution x . Often **only the modified components of x determine**

- the **feasibility** of the **new subset $o(x)$**
- the **variation of the objective function** $\delta f_o(x) = f(o(x)) - f(x)$

Then, **two operations** $o, o' \in \mathcal{O}$ that **modify different components of x**

- are **compatible and commutable**

$$o'(o(x)) = o(o'(x)) \in X$$

- have an **overall effect independent of the order of application** and easy to compute: for additive functions it is usually the sum

$$\delta f_{oo'}(x) = \delta f_{o'o}(x) = \delta f_o(x) + \delta f_{o'}(x)$$

The idea is to **perform a whole set of moves combining their effects**.

6.8.2 Dynasearch

Let a **composite move** be a **set of elementary moves** with mutually independent effects on feasibility and the objective.

The situation can be modeled with an **improvement matrix** A in which

- the **rows** represent the **components of the solution** (e.g., branches in the CMSTP, circuits in the VRP, circuit segments in the TSP)
- the **columns** represent the **elementary moves** $o \in \mathcal{O}$ and the **value of a column** equals the **objective improvement** $-\delta f_o(x)$
- $a_{io} = 1$ when **move o affects component i** , $a_{io} = 0$ otherwise

Determine an **optimum packing of the columns**, that is a subset of non-conflicting columns of maximum value.

The Set Packing Problem is in general \mathcal{NP} -hard, but

- on special matrices it is polynomial (as in the matrix from the TSP)
- if each move modifies at most two components
 - the rows can be seen as vertices of a graph
 - the columns can be seen as edges of a graph
 - each packing of columns becomes a matchingand the maximum matching problem is polynomial

6.8.3 Cyclic exchanges

Another set of exponential neighborhood that can be explored in polynomial time.

In many problems

- A **feasible solution** is a **partition of objects into components** $S^{(\ell)}$, that is an assignment of objects to components (i, S_i) (vertices or edges into branches for the CMSTP, nodes or arcs into circuits for the VRP, objects into containers in the BPP, etc.; the ground set is a cartesian product of vertices and subtrees, objects and containers, ecc.; a single solution is a partition).
- The **feasibility is associated to the single components** (the feasibility can be checked against the single component, such as container, subtree, ecc.).
- The **objective function is additive with respect to the components**

$$f(x) = \sum_{\ell=1}^r f(S^{(\ell)})$$

In these problems, it is natural to define the **set of operations** \mathcal{T}_k which includes the **transfers of k elements from their component to another** and to derive from \mathcal{T}_k the **neighborhood** $N_{\mathcal{T}_k}$

- Often the feasibility constraints forbid the simple transfers ($N_{\mathcal{T}_1}$ is often unfeasible, if all containers are full I can't simply put an object into another container).
- But the number of multiple transfers quickly grows with k .

We want to **find** a **subset of** $N_{\mathcal{T}_k}$ **large**, but **efficient** to explore.

6.8.4 The improvement graph

Allows to describe sequences of transfers

- A **node** i corresponds to an **element** i of the **current solution** x .
- An **arc** (i, j) corresponds to
 - the **transfer** of element i from its current **component** S_i to the current **component** S_j of element j
 - the **deletion** of element j from **component** S_j
- The **cost of arc** c_{ij} corresponds to the (positive or negative) **variation of the contribution of** S_j to the objective

$$c_{ij} = f(S_j \cup \{i\} \setminus \{j\}) - f(S_j)$$

with $c_{ij} = +\infty$ if it is **unfeasible** to replace j with i in S_j

A circuit in such a graph corresponds to a closed sequence of transfers.

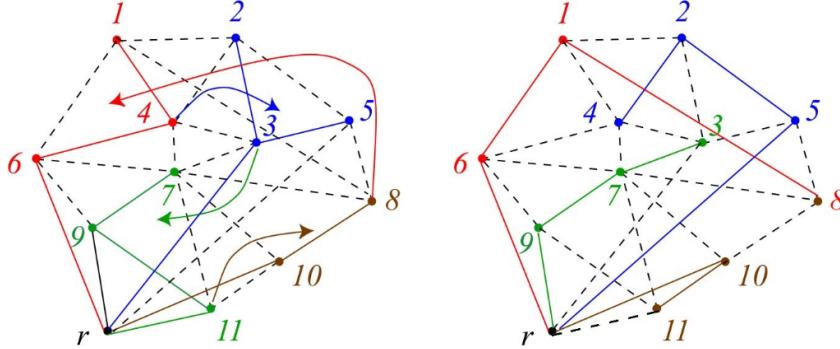
If I put i in place of j , j has to go somewhere, for example in the place of k , and so on, until I have a feasible sequence of transfer, i.e. a cycle in the improvement graph (each move must belong to a different component, otherwise problems).

The **cost of the circuit** corresponds to the **cost of the sequence**

- but only if each node belongs to a different component

Find the **minimum cost circuit** satisfying this condition.

Example: the CMSTP



Consider the composite move $(4, 3), (3, 11), (11, 8), (8, 4)$:

- 4 moves into the blue branch to replace 3
- 3 moves into the green branch to replace 11
- 11 moves into the brown branch to replace 8
- 8 moves into the red branch to replace 4

The cost variation for subtree S_j yields the cost of arc c_{ij} .

The weight of branch S_j varies by $w_i - w_j$: if unfeasible, forbid the arc. The feasibility is determined by the single transfer, if the new cost is still feasible the transfer is feasible.

The total cost is the sum of all variation due to the transfers made.

If you visit two elements in a single component it is not guaranteed that the feasibility depends only on the single transfer.

Search for the minimum cost circuit: The problem is actually \mathcal{NP} -hard, but

- The constraint of visiting only once each component allows a rather efficient **dynamic programming algorithm** that grows partial paths (if the components are r , the circuit has at most r arcs).

- All **partial paths of cost ≥ 0 can be neglected** because
 - the total variation of the objective sums the effect of the single moves

$$\delta f_{o_1, \dots, o_k}(x) = \sum_{\ell=1}^k \delta f_{o_\ell}(x)$$

- every **sequence of numbers with negative sum** admits a cyclic permutation **whose partial sums are all negative**. e.g., $(+1, -2, +4, -10, +2)$ admits $(-10, +2, +1, -2, +4)$ (if the total is negative I can start with a big negative and stay in the negative all the way)
- therefore, **there is a cyclic permutation of the moves o_1, \dots, o_k**

$$\delta f_{o_1, \dots, o_k}(x) < 0 \implies \exists h : \delta f_{o_{(h+1) \bmod k}, \dots, o_{(h+\ell) \bmod k}}(x) < 0 \text{ for } \ell = 1, \dots, k$$

that is, **improving at each step**.

Moreover,

- There are **heuristic polynomial algorithms** for the problem.
- There are **polynomial algorithms to solve relaxations** of the problem that neglect the constraint on the components, finding
 - a nonminimum negative circuit (Floyd-Warshall), if any exists
 - a circuit of minimum average cost (total cost divided by number of arcs)

If the cost of such relaxed solutions is

- positive, then no negative circuit exists
- negative, then the relaxed solution can be
 - * optimal (if luckily they are feasible)
 - * a starting point to generate a feasible heuristic solution

Noncyclic exchange chains: It is also possible to create noncyclic transfer chains, so that the cardinality of the components can vary.

It is enough to add to the improvement graph

- A **source node** (fake, doesn't represent any node of the problem).
- A **node for each component** (a fake node for each component in the solution, for example each subtree).
- **Arcs** from the **source node** to the **nodes associated to the elements**.
- **Arcs** from the **nodes associated to the elements** to the **nodes associated to the components**.

Then, find the minimum cost path that

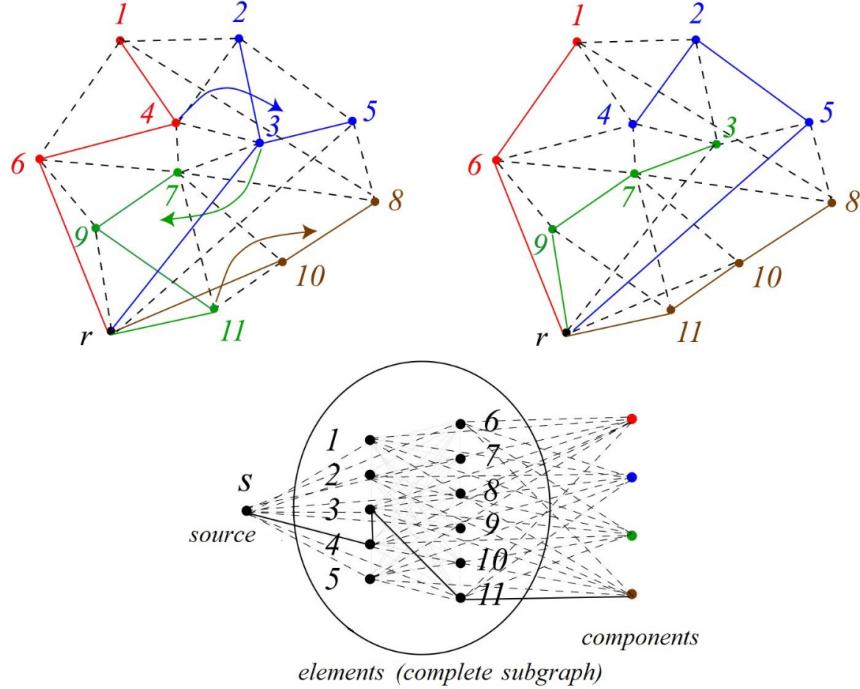
- **starts** from the **source node**
- **ends** in a **component node**
- **visits at most one node for each component**

These paths correspond to **open transfer chains** in which

- a component loses an element
- zero or more components lose an element and acquire another one
- a component acquires an element

You just get an element from a source node, swap n elements among components, end in a component which does not lose an element.

Example: the CMSTP



Noncyclic exchange $(s, 4), (4, 3), (3, 11), (11, S_4)$.

4 goes to s (it just means that we remove it), then we swap 4 and 3, and so on until in the green subtree we swap 11, which goes to brown, with S_4 (another fake element).

It's just a notation to denote empty first and last swaps.

6.9 Order-first split-second

The **Order-first split-second** method for **partition problems**

- Builds a **starting permutation of the elements** to be partitioned.
- **Partitions the elements into components in an optimum way** under the additional constraint that **elements of the same component be consecutive in the starting permutation**.

Of course, the solution depends on the starting permutation: it is reasonable to repeat the resolution for different permutations creating a two-level method

1. the upper level selects a permutation
2. the lower level computes the optimal partition for the permutation

Problem: different permutations yield the same solution (the permutations are more numerous than the solutions).

The auxiliary graph: Once again, we exploit an auxiliary graph.

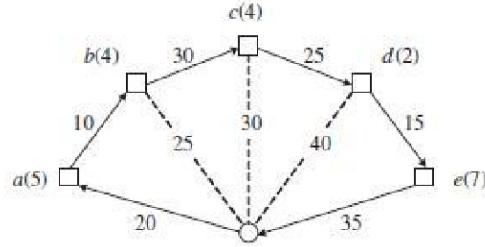
Given the permutation (s_1, \dots, s_n) of the elements

- each **node** v_i corresponds to an element s_i plus a fictitious node v_0
- each **arc** (v_i, v_j) with $i < j$ corresponds to a **potential component** S_ℓ that assigns to the same subset the elements (s_{i+1}, \dots, s_j)
 - from s_i excluded
 - to s_j included
- the **cost** c_{v_i, v_j} corresponds to the **cost of the component** $f(S_\ell)$
- the **arc does not exist** if the **component is unfeasible**

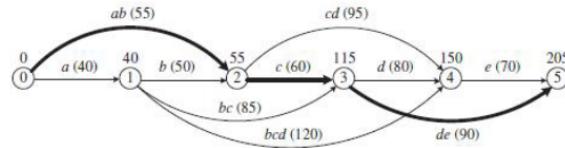
Consequently

- each path from v_0 to v_n represents a solution (partition of elements)
- the cost of the path coincides with the cost of the partition
- the graph is **acyclic: finding the optimum path costs $O(m)$** where $m \leq n(n - 1)/2$ is the number of arcs

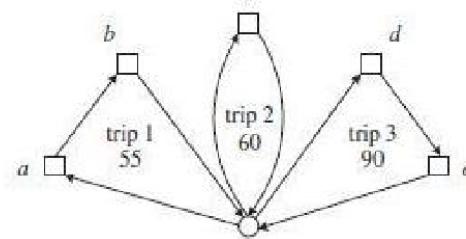
Example: the VRP. Given an instance of VRP with 5 nodes and capacity $W = 10$



the arcs corresponding to unfeasible paths (weight $> W$) do not exist, the costs of the arcs are the costs of the TSP solutions for $\{d, v_{i+1}, \dots, v_j\}$



The optimal path corresponds to three circuits: $(d, v1, v2, d)$, $(d, v3, d)$ and $(d, v4, v5, d)$



6.10 Variable Depth Search (VDS)

In the VDS a **composite move** is a **sequence of elementary moves**

- Consider each solution x' in the basic neighborhood $N_{\mathcal{O}_1}(x)$.
- From it, make a sequence of moves **optimizing each elementary step**, but **allowing worsening moves** and **forbidding backward moves**.
- **Terminate** when the **current solution** y becomes **worse than x'** or all moves are **forbidden** (the length k of the sequence is variable).
- **Return the best solution y^* found** along the sequence.

Scheme of the VDS: Given $x^{(t)}$, for each $x' \in N(x^{(t)})$, instead of evaluating only $f(x')$

1. Find a promising solution \tilde{y} in a neighborhood $\hat{N}(x') \subseteq N(x')$.
2. As long as \tilde{y} improves $x^{(t)}$, replace x' with \tilde{y} and go to 1.
3. Return the best solution y^* found during the whole process.

For each $x' \in N(x)$

Algorithm 8 Steepest Descent

Compute $f(x')$

Algorithm 9 Variable Depth Search

```
y := x'; y* := x'; Stop := false;  
while Stop = false do  
     $\tilde{y} := \arg \min_{y' \in \hat{N}(y)} f(y')$   
    if  $f(\tilde{y}) \geq f(x')$  then  
        Stop := true  
    else  
        y :=  $\tilde{y}$   
    end if  
    if  $f(\tilde{y}) < f(y^*)$  then  
         $y^* := \tilde{y}$   
    end if  
end while  
return  $f(y^*)$ ;
```

It is a sort of roll-out mechanism for exchange algorithms.

Instead of just computing the value of the function in x' , you try to “launch” a local search heuristic; starting from x' you repeatedly explore a neighborhood of the current solution \tilde{y} , as long as you don’t satisfy a termination condition, keeping track of the best solution found all along.

You get a candidate solution and then you explore, starting from that solution, a (much) smaller neighborhood (to have a fast exploration) and repeat this cycle until a termination condition holds, i.e. when the new solution is worse than the original one.

By accepting worsening moves (with a bound in the original solution) it allows to go further in the search space than a simple steepest descent.

Differences to steepest descent: With respect to steepest descent exploration

- VDS **finds a local optimum for each solution of the neighborhood** performing a sort of one-step look-ahead.
- VDS **admits worsenings along the sequence** of elementary moves (but never with respect to the starting solution).
- VDS **makes moves that increase the distance from the starting point** to avoid cyclic behaviors (gradually restricting the neighborhood).

In order to limit the computational effort

- the **elementary moves** use a **reduced neighborhood** $\hat{N} \subseteq N$
- \hat{N} (elementary step) is **explored** with the **first-best strategy** (as soon as you find an improvement you take it)
- N (basic neighborhood) is **explored** with the **first-best strategy**

Instead of just computing the objective function you're trying to “have a look” far away.

6.10.1 Lin-Kernighan's algorithm for the symmetric TSP

Still the best performing algorithm for the TSP (a variation of it).

Neighborhood $N_{\mathcal{R}_k}(x)$ includes the solutions obtained

- deleting k arcs of x
- adding other k arcs that recreate a Hamiltonian circuit
- possibly inverting parts of the circuit (leaving the cost unchanged)

Lin-Kernighan's algorithm is a VDS with **sequences of 2-opt exchanges**: a k -opt exchange is equivalent to a sequence of $(k - 1)$ 2-opt exchanges, where each deletes one of the two arcs added by the previous exchange.

Then for each solution $x' \in N_{\mathcal{R}_2}(x)$ obtained by exchange (i, j)

- evaluate the 2-opt exchanges that delete the added arc (s_i, s_{j+1}) and each arc of $x \cap x'$ to find the best exchange (i', j')
- if this improves upon x , perform exchange (i', j') , obtaining x''
- evaluate the exchanges that delete $(s_{i'}, s_{j'+1})$ and each arc of $x \cap x''$...
- ...
- if the best solution among x', x'', \dots is better than x , accept it

It essentially does a 2-opt exchange among 2 arcs and then deletes one (fixed) and takes time $O(n)$ to search the best possible exchange across all other nodes in the solution. Reverse the path among nodes when needed. Example not provided, too long (L14 slides).

The arc to be removed is never the one chosen after the search, always the other one, it would be a backward move otherwise.

Implementation details:

- **Each step deletes an arc of the starting solution** to avoid going back and one of the arcs added in the previous step to reduce complexity.
- This imposes an **upper bound on the length of the sequence**.
- **Stopping the sequence as soon as the solution is no longer better than the starting solution does not impair the result**
 - the total variation of the objective sums the effect of the single moves
 - every sequence of numbers with negative sum admits a cyclic permutation whose partial sums are all negative
 - therefore, there is a cyclic permutation of the moves o_1, \dots, o_k that is, improving at each step (these last two points have already been discussed)

6.11 Iterated greedy methods (destroy-and-repair)

Every **exchange** can be seen as a **combination of addition and deletion**

$$x' = x \cup A \setminus D$$

with $A = x' \setminus x$ and $D = x \setminus x'$.

However

- single swaps $x' = x \cup \{j\} \setminus \{i\}$ can give bad or unfeasible results
- larger neighborhoods can be inefficient
- in many problems the right cardinalities of A and D are unknown, because the solutions have nonuniform cardinality (e.g., KP, SCP. . .)

A possible idea is to

1. delete from x a subset $D \subset x$ of cardinality $\leq k$ (destroy heuristic)
2. complete it with a constructive heuristic (repair heuristic)

or, of course, **the opposite**

1. add to x a set $A \subset B \setminus x$ of cardinality $\leq k$
2. reduce it with a destructive heuristic

Selection of A and D : Most of the time both subsets are chosen heuristically, not exhaustively

- tuning their size $|A|$ and $|D|$ with some parameter
- selecting promising elements based on their cost/value
- applying the first-best strategy (immediately accept any improving solution)

Usually both subsets are chosen in a randomized way (metaheuristics then).

6.12 Overcoming local optima

The steepest descent exchange heuristics only provide local optima.

In order to **improve**, one can

- **repeat** the search (How to avoid following the same path?)
- **extend** the search (How to avoid falling in the same optimum? If I start in a neighborhood of my local optimum chances are I'll fall back into it)

In the constructive algorithms only repetition was possible.

The constructive metaheuristics exploit **randomization** and **memory** to operate on $\Delta_A^+(x)$ and $\varphi_A(i, x)$.

The **exchange metaheuristics** exploit them to operate on

- the **starting solution** $x^{(0)}$ (multi-start, ILS, VNS)
- the **neighborhood** $N(x)$ (VND)
- the **selection criteria** $\varphi(x, A, D)$ (DLS/GLS)
- the **selection rule** $\arg \min$ (SA, TS)

6.12.1 Termination condition

A search that repeats or extends beyond a local optimum can ideally be infinite. If you prolong the search, when does it end?

In practice, one uses **termination conditions** that can be “**absolute**”

1. A given **total number of explorations of the neighborhood** or a given **total number of repetitions of the local search**.
2. A given **total execution time**.
3. A given **value of the objective**.

or “**relative**” to the **profile of f^***

1. A given **number of explorations** of the neighborhood or repetitions **after the last improvement of f^*** .
2. A given **execution time after the last improvement**.
3. A given **minimum value of the ratio between improvement of the objective and number of explorations/repetitions or execution time** (e.g.: f^* improves less than 1% in the last 1000 explorations).

Fair comparisons require **absolute conditions** (give both the same time/number of exploration).

6.13 Modify the starting solution

It is possible to create **different starting solutions**

- Generating them at **random**
 - with uniform probability
 - with biased distributions (based on the data, possibly on memory)
- Applying different **constructive algorithms**
 - heuristics
 - metaheuristics (with randomization and/or memory)
- Applying the **exchange algorithm to modify the solutions visited** (therefore with memory, and usually also randomization)

6.13.1 Random generation

The **advantages** of random generation are

- Conceptual **simplicity**.
- **Quickness** for the problems in which it is easy to guarantee feasibility.
- **Control** on the **probability distribution** in X based on
 - element cost (e.g., favor the cheapest elements)
 - element frequency during the past search, to favor the most frequent elements (intensification) or the less frequent ones (diversification). This combines randomization and memory
- **Asymptotic convergence to the optimum** (in infinite time).

The **disadvantages** of random generation are

- **Scarce quality** of the **starting solutions** (not the final ones).
- **Long times** before **reaching the local optimum**. This depends on the complexity of the exchange algorithm.
- **Inefficiency** when deciding feasibility is \mathcal{NP} -complete.

6.13.2 Constructive procedures

Multi-start methods are the classical approach

- design **several constructive heuristics**
- **each constructive heuristic generates a starting solution**
- **each starting solution is improved by the exchange heuristic**

The disadvantages are

1. **scarce control**: the generated solutions tend to be similar
2. **impossibility to proceed indefinitely**: the number of repetitions is fixed
3. **high design effort**: several different algorithms must be designed
4. **no guarantee of convergence**, not even in infinite time

Consequently, constructive metaheuristics are preferred nowadays. The initial step is given by a constructive metaheuristic.

GRASP and Ant System include by definition an exchange procedure.

What's the difference between a metaheuristic with a randomized/memory based constructive phase and an exchange heuristic with initialized by a randomized process? There's little difference.

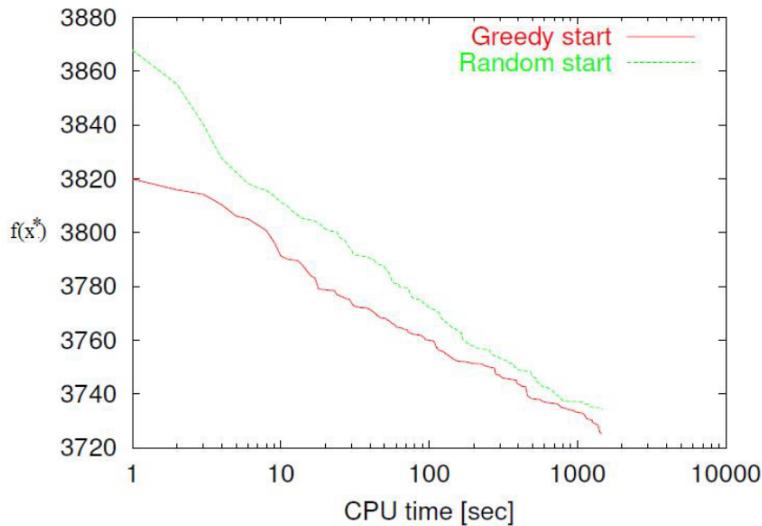
6.13.3 Influence of the starting solution

The starting solution obviously has influence on the algorithm's performance, so how do we determine if we can choose a random generation or if we need a good constructive heuristic?

If the **exchange heuristic** is

- **Good**, the starting solution has a **short-lived influence**: a random or heuristic generation of $x^{(0)}$ are very similar.
- **Bad**, the starting solution has a **long-lived influence**: a good heuristic to generate $x^{(0)}$ is useful.

If the heuristic is not good, it might be a good idea to spend some time generating the starting solution, while if the exchange phase is very good the starting solution doesn't matter as much.



This exchange heuristic is not very good, there's still a perceivable (albeit small) difference at the end, after a significant amount of time (note the logarithmic scale).

6.13.4 Exploiting the previous solutions

The idea is to **exploit the information on previously visited solutions**

- save **reference solutions**, such as the best local optimum found so far and possibly other local optima
- generate the new starting solution modifying the reference ones

During the exchange heuristic you save some solutions that are “interesting” (the best, possibly others, usually good-quality solutions) and generate some other starting solutions based on these.

The advantages of this approach are

- **control**: the modification can be reduced or increased *ad libitum*. You can intensify or diversify the search at will (in respect to the starting point)
- **good quality**: the starting solution is very good (you start from a good-quality solution)
- **conceptual simplicity**: just design a modification, no need to invent much new stuff
- **implementation simplicity**: the modification can be performed with the operations defining the neighborhood
- **asymptotic convergence to the optimum** under suitable conditions

6.14 Iterated Local Search (ILS)

The Iterated Local Search (ILS), proposed by Lourenço, Martin and Stützle (2003) requires

- a **steepest descent** exchange heuristic to **produce local optima**
- a **perturbation procedure** to **generate the starting solutions**
- an **acceptance condition** to decide whether to **change the reference solution x**
- a **termination condition**

Algorithm 10 Algorithm *IteratedLocalSearch*($I, x^{(0)}$)

```
 $x := \text{SteepestDescent}(x^{(0)})$ 
 $x^* := x$ 
for  $l := 1$  to  $\ell$  do
     $x' := \text{Perturbate}(x)$ 
     $x' := \text{SteepestDescent}(x')$ 
    if  $\text{Accept}(x', x^*)$  then
         $x := x'$ 
    end if
    if  $f(x') < f(x^*)$  then
         $x^* := x'$ 
    end if
end for
return  $(x^*, f(x^*))$ 
```

It iteratively applies local search by generating new starting solutions every time, so you need an exchange heuristic on which to base this on.

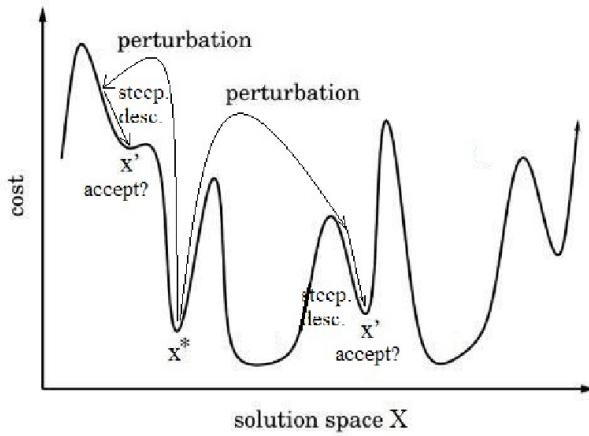
The perturbation procedure is used to generate new starting solutions from the reference.

The acceptance condition is needed to decide if the local optimum found can be accepted as the new reference (otherwise the old one is kept and “perturbed” for a new starting solution).

In the end it returns the best solution found.

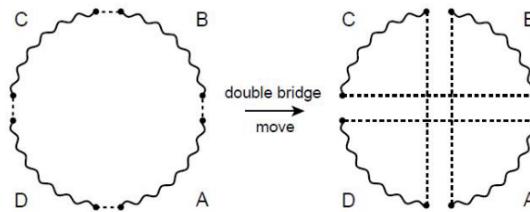
The idea is that

- The **exchange heuristic** quickly explores an attraction basin, terminating into a local optimum.
- The **perturbation procedure moves to another attraction basin.**
- The **acceptance condition evaluates if the new local optimum is a promising starting point** for the following perturbation.



Example: A classical application of ILS to the TSP uses

- exchange heuristic: steepest descent with neighborhood $N_{\mathcal{R}_2}$ (remove 2 edges)
- perturbation procedure: a double-bridge move that is particular kind of 4-exchange



- acceptance condition: the best known solution improves

$$f(x') < f(x^*)$$

The reference solution is the best known one ($x = x^*$)

6.14.1 Perturbation procedure

Let \mathcal{O} be the operation set that defines neighborhood $N_{\mathcal{O}}$.

The **perturbation procedure** performs a **random operation** o

- with $o \in \mathcal{O}' \not\subseteq \mathcal{O}$, to avoid the exchange heuristic driving solution x' back to the starting local optimum x (you need a new starting point, go far enough to get one)

Two typical definitions of \mathcal{O}' are

- **sequences of $k > 1$ operations of \mathcal{O}** (generating a random sequence is cheap)
- **conceptually different operations** (e.g., vertex exchanges instead of edge exchanges)

The main difficulty of ILS is in tuning the perturbation: if it is

- **Too strong**, it turns the search into a **random restart** (if you go way too far it's just a new random start).
- **Too weak**, it guides the search back to the **starting local optimum** (you are too close to the previous one)
 - wasting time
 - possibly losing the asymptotic convergence

Ideally one would like to **enter any basin** and **get out of any basin** (and the perturbation must be tuned according to the specific basin).

6.14.2 Acceptance condition

The acceptance condition balances intensification and diversification

- Accepting **only improving** solutions favors **intensification**

$$\text{Accept}(x', x^*) := (f(x') < f(x^*))$$

The reference solution is always the best found: $x = x^*$.

- Accepting **any** solution favors **diversification**

$$\text{Accept}(x', x^*) := \text{true}$$

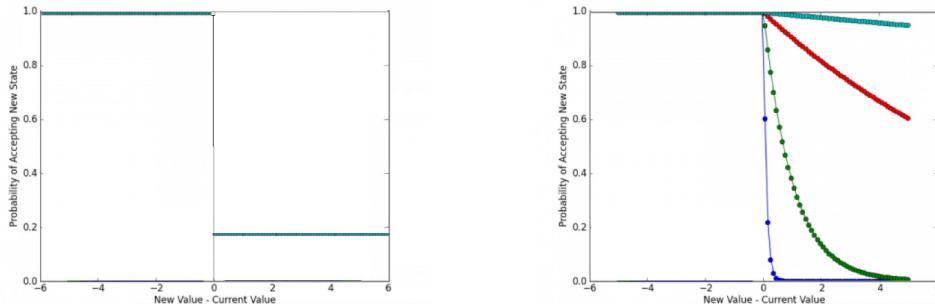
The reference solution is always the last optimum found: $x = x'$.

Intermediate strategies can be defined **based on** $\delta f = f(x') - f(x^*)$ the difference in the objective function

- if $\delta f < 0$, **always accept** x'
- if $\delta f \geq 0$, **accept** x' with **probability** $\pi(\delta f)$, where $\pi(\cdot)$ is a nonincreasing function

The most **typical** cases are:

- **constant probability**: $\pi(\delta f) = \bar{\pi} \in (0; 1)$ for each $\delta f \geq 0$
- **monotonically decreasing probability** with $\pi(0) = 1$ and $\lim_{\delta f \rightarrow +\infty} \pi(\delta) = 0$; the probability decreases proportionally to the worsening of the solution



Memory can also be used, accepting x' more easily if many iterations have elapsed since the last improvement of x^* .

6.15 Variable Neighborhood Search (VNS)

A method very similar to ILS is the Variable Neighborhood Search proposed by Hansen and Mladenović (1997).

The **main differences** between ILS and VNS are the **use of**

- The **strict acceptance condition**: $f(x') < f(x^*)$ (only intensifies, simpler).
- An **adaptive perturbation mechanism** instead of the fixed one (more complicated than ILS).

VNS often introduces also neighborhood modifications (later on this).

The **perturbation mechanism** is based on a **hierarchy of neighborhoods**, that is a **family of neighborhoods with an increasing parametric size s**

$$N_1 \subset N_2 \subset \dots \subset N_s \subset \dots N_{s_{max}}$$

Typically the parameterized neighborhoods used are

- $N_{\mathcal{H}_s}$, based on the Hamming distance between subsets
- $N_{\mathcal{O}_s}$, based on the sequences of operations from a basic set \mathcal{O}

and $x^{(0)}$ is extracted randomly from a neighborhood of the hierarchy.

Algorithm 11 Algorithm $VariableNeighbourhoodSearch(I, x^{(0)}, s_{min}, s_{max}, \delta s)$

```
x := SteepestDescent(x(0))
x* := x
s := smin
for l := 1 to ℓ do
    x' := Shaking(x*, s)
    x' := SteepestDescent(x')
    if f(x') < f(x*) then
        x* := x'
        s := smin
    else
        s := s + δs
    end if
    if s > smax then
        s := smin
    end if
end for
return (x*, f(x*))
```

- the **reference solution** x' is always the best known solution x^*
- the **starting solution** is obtained extracting it at **random from the current neighborhood of the reference solution** $N_s(x^*)$
- the **exchange heuristic** produces a **local optimum** with respect to the **basic neighborhood** N
- if the **best known solution improves**, the **current neighborhood becomes** $N_{s_{min}}$
- **otherwise, move to a larger neighborhood** $N_{s+\delta s}$, never exceeding $N_{s_{max}}$

6.15.1 Adaptive perturbation mechanism

It is called **variable neighborhood** because the **neighborhood used to extract $x^{(0)}$ varies** based on the results of the exchange heuristic

- **if a better solution is found**, use the **smallest neighborhood**, to generate a starting solution very close to x^* (intensification)
- **if a worse solution is found**, use a **slightly larger neighborhood**, to generate a starting solution slightly farther from x^* (diversification)

If you find a better solution you accept it and set s to the minimum possible value, the new starting solution is close to the current one (you get closer, perturb only a little).

If you find a worse solution you perturbed too little, you need to increase s and perturb more (starting solution that is farther away from the current one).

The method has **three parameters**

1. s_{min} identifies the **smallest neighborhood to generate new solutions**
2. s_{max} identifies the **largest neighborhood to generate new solutions**
3. δ_s is the **increase of s between two subsequent attempts** (how much does the neighborhood increase when it fails?)

The exchange heuristic adopts a **small neighborhood to be efficient** (N_1 , or anyway N_s with $s \leq s_{min}$).

Tuning of the shaking parameters: The value of s_{min} must be

- Large enough to get out of the current attraction basin.
- Small enough to avoid jumping over the adjacent attraction basins.

In general, one sets $s_{min} = 1$, and increases it if experimentally profitable.

The value of s_{max} must be

- Large enough to reach any useful attraction basin.
- Small enough to avoid reaching useless regions of the solution space.

Example: the diameter of the search space for the basic neighborhood: $\min(k, n - k)$ for the MDP; n for the TSP and MAX-SAT, etc.

The value of δs must be

- Large enough to reach s_{max} in a reasonable time.
- Small enough to allow each reasonable value of s .

In general, one sets $\delta s = 1$, unless $s_{max} - s_{min}$ is too large.

6.15.2 Skewed VNS

In order to **favor diversification**, it is possible to **accept x' when**

$$f(x') < f(x^*) + \alpha d_H(x', x^*)$$

where

- $d_H(x', x^*)$ is the **Hamming distance** between x' and x^*
- $\alpha > 0$ is a **suitable parameter**

Accept bounded worsening solution, if they're far enough away (measured with the Hamming distance). How distant they are determines how much worse they can get.

This allows to **accept worsening solutions** as long as they are **far away**

- $\alpha \approx 0$ tends to accept only improving solutions
- $\alpha \gg 0$ tends to accept any solution

The parameter α is the “worsening bound”, determines how much worse the solution can get and still get accepted (if it's far enough away).

Of course, the random strategies seen for the ILS can also be adopted.

6.16 Extending the local search without worsening

Instead of repeating the local search, extend it beyond the local optimum.

To avoid worsening solutions, the selection step must be modified

$$\tilde{x} := \arg \min_{x' \in N(x)} f(x')$$

And two main strategies allow doing that

- The **Variable Neighborhood Descent** (VND), **changes the neighborhood N**
 - it guarantees an evolution with no cycles (the objective improves)
 - it terminates when all neighborhoods have been exploited
- The **Dynamic Local Search** (DLS) **changes the objective function f** (\tilde{x} is better than x for the new objective, possibly worse for the old)
 - it can be trapped in loops (the new objective changes over time)
 - it can proceed indefinitely

6.17 Variable Neighborhood Descent (VND)

The Variable Neighborhood Descent of Hansen and Mladenović (1997) exploits the fact that a **solution is locally optimal for a specific neighborhood**

- a **local optimum** can be **improved** using a **different neighborhood**

If you change the neighborhood, the local optimum changes.

Given a **family of neighborhoods** $N_1, \dots, N_{s_{tot}}$

1. set $s := 1$
2. apply a **steepest descent exchange heuristic** and find a **local optimum \bar{x} with respect to N_s**
3. **flag all neighborhoods** for which \bar{x} is **locally optimal** and update s
4. if \bar{x} is a local optimum for all N_s , terminate; otherwise, go back to point 2

Algorithm 12 Algorithm $VariableNeighbourhoodDescent(I, x^{(0)})$

```

 $flag_s := false \forall k$ 
 $\bar{x} := x^{(0)}$ 
 $x^* := x^{(0)}$ 
 $s := 1$ 
while  $\exists s : flag_s = false$  do
     $\bar{x} := SteepestDescent(\bar{x}, s)$  // possibly truncated
     $flag_s := true$ 
    if  $f(\bar{x}) < f(x^*)$  then
         $x^* := \bar{x}$ 
         $flag_{s'} := false \forall s' \neq s$ 
    end if
     $s := Update(s)$ 
end while
return  $(x^*, f(x^*))$ 

```

Anticipated termination of Steepest Descent: Using many neighborhoods means that some might be rather **large** and **slow to explore**.

In order to increase the efficiency of the method one can

- Adopt a **first-best strategy in the larger neighborhoods**.
- **Terminate the Steepest Descent before reaching a local optimum** (possibly even after a single step).

Larger neighborhoods aim to **move out of the basins of attraction of smaller ones**.

VND and VNS: There is of course a strict relation between VND and VNS (in fact, they were proposed in the same paper).

The **fundamental differences** are that in the VND

- At each step the **current solution is the best known one**.
- The **neighborhoods are explored**, instead of being used to extract random solutions; they are never huge.
- The **neighborhoods do not necessarily form a hierarchy**; the update of s is not always an increment.
- When a **local optimum for each N_s has been reached, terminate**; VND is deterministic and would not find anything else.

6.17.1 Neighborhood update strategies for the VND

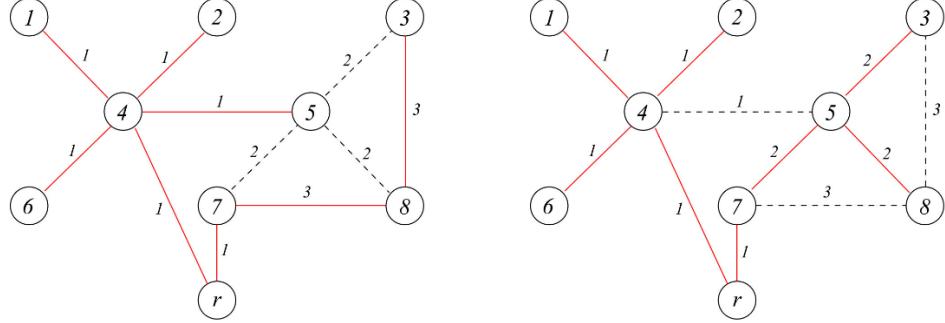
There are two main classes of VND methods

- Methods with **heterogeneous neighborhoods**
 - exploit the potential of **topologically different neighborhoods**
(e.g., exchange vertices instead of edges)
- Consequently, s periodically scans the values from 1 to s_{tot} (possibly randomly permuting the sequence at each repetition).
- Methods with **hierarchical neighborhoods** ($N_1 \subset \dots \subset N_{s_{tot}}$)
 - **fully exploit the small and fast neighborhoods**
 - resort to the **large and slow ones only to get out of local optima** (usually terminating Steepest Descent prematurely)
- Consequently, the update of s works as in the VNS
 - when **no improvements** can be found in N_s , **increase s**
 - when **improvements** can be found in N_s , **decrease s back to 1**

Terminate when the current solution is a local optimum for all N_s

- in the **heterogeneous case**, terminate when **all fail**
- in the **hierarchical case**, terminate when the **largest fails**

Example: This instance of CMSTP has $n = 9$ vertices, uniform weights ($w_v = 1$), capacity $W = 5$ and the reported costs (graph is complete but the missing edges have $c_e \gg 3$, so they are not drawn).



Consider neighborhood N_{S_1} (single-edge swaps) for the first solution:

- no edge in the right branch can be deleted because the left branch has zero residual capacity and a direct connection to the root would increase the cost
- deleting any edge in the left branch increases the total cost, the solution is a local optimum for N_{S_1}

Neighborhood N_{T_1} (single-vertex transfers) has an improving solution, obtained moving vertex 5 from the left branch to the right one.

On the left the solution for N_{S_1} , on the right the one for N_{T_1} (obviously the case of heterogeneous neighborhoods).

6.18 Dynamic Local Search (DLS)

The Dynamic Local Search is also known as Guided Local Search.

Its approach is **complementary** to VND

- it **keeps the starting neighborhood**
- it **modifies the objective function**

It is often used when the **objective is useless** because it has **wide plateaus** (e.g. in the PMSP the completion time is often the same, even after multiple movements).

The **basic idea** is to

- Define a **penalty function** $w : X \rightarrow \mathbb{N}$ (defined on the solution, usually integer values).
- Build an **auxiliary function** $\tilde{f}(f(x), w(x))$ which **combines the objective function f with the penalty w** .
- Apply a **steepest descent exchange heuristic to optimize \tilde{f}** .
- At each iteration **update the penalty w based on the results** (and consequently update the function \tilde{f} and the landscape of the search).

The penalty is adaptive in order to move away from recent local optima but this introduces the risk of cycling.

Algorithm 13 Algorithm *DynamicLocalSearch*($I, x^{(0)}$)

```
w := StartingPenalty(I)
x̄ := x(0)
x* := x(0)
while Stop() = false do
    (x̄, xf) := SteepestDescent(x̄, f, w) // possibly truncated
    if f(xf) < f(x*) then
        x* := xf
    end if
    w := UpdatePenalty(w, x̄, x*)
end while
return (x*, f(x*))
```

Notice that the steepest descent heuristic

- **Optimizes a combination \tilde{f} of f and w** (Steepest Descent is applied to both).
- **Returns two solutions:**
 1. a final solution \bar{x} , locally **optimal** with respect to \tilde{f} , to update w
 2. a **solution** x_f , that is the **best it has found with respect to f**

6.18.1 Variants

The **penalty can be applied** (for example)

- **Additively** to the elements of the solution:

$$\tilde{f}(x) = f(x) + \sum_{i \in x} w_i$$

The function becomes the sum of the objective with the sum of the penalties for all elements in the solution; each element is penalized, it searches for elements with low penalty.

- **Multiplicatively** to components of the objective $f(x) = \sum_j \phi_j(x)$:

$$\tilde{f}(x) = \sum_j w_j \phi_j(x)$$

The objective function is defined in terms of components and each one is multiplied by its penalty.

The **penalty can be updated**

- at **each single neighborhood exploration**
- when a **local optimum for \tilde{f}** is reached
- when the **best known solution x^*** is unchanged for a long time

The **penalty can be modified** with

- **Random updates**: “noisy” perturbation of the costs.
- **Memory-based updates**, favoring the most frequent elements (intensification) or the less frequent ones (diversification).

These two are not the only ways, they are just examples, more can be defined.

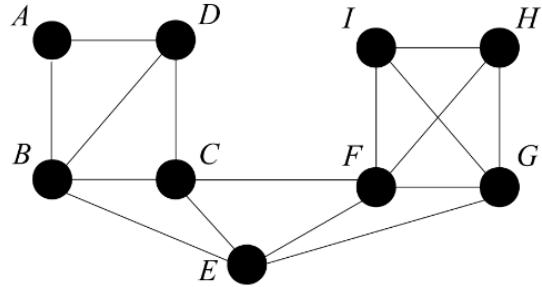
Example: DLS for the MCP. Given an undirected graph, find a maximum cardinality clique (subset of vertices that are fully connected)

- The exchange heuristic is a VND using the neighborhoods
 1. N_{A_1} (vertex addition): the solution always improves, but the neighborhood is very small and often empty
 2. N_{S_1} (exchange of an internal vertex with an external one): the neighborhood is larger, but forms a plateau (uniform objective)

- The objective provides no useful direction in either neighborhood.
- Associate to each vertex i a penalty w_i initially equal to zero.
- The exchange heuristic minimizes the total penalty (within the neighborhood).
- Update the penalty
 1. when the exploration of N_{S_1} terminates: the penalty of the current clique vertices increases by 1
 2. after a given number of explorations: all the nonzero penalties decrease by 1

The rationale of the method consists in aiming to

- expel the internal vertices (diversification)
- in particular, the oldest internal vertices (memory)



Start from $x^{(0)} = \{B, C, D\}$, with $w = [0 1 1 1 0 0 0 0 0]$

1. $w(\{B, C, E\}) = w(\{A, B, D\}) = 2$, but $\{A, B, D\}$ wins lexicographically: $x^{(1)} = \{A, B, D\}$ with $w = [1 2 1 2 0 0 0 0 0]$
2. $x^{(2)} = \{B, C, D\}$ with $w = [1 3 2 3 0 0 0 0 0]$ is the only neighbor
3. $w(\{B, C, E\}) = 5 < 7 = w(\{A, B, D\})$: $x^{(3)} = \{B, C, E\}$ with $w = [1 4 3 3 1 0 0 0 0]$
4. $w(\{C, E, F\}) = 4 < 10 = w(\{B, C, D\})$: $x^{(4)} = \{C, E, F\}$ with $w = [1 4 4 3 2 1 0 0 0]$
5. $w(\{E, F, G\}) = 3 < 11 = w(\{B, C, E\})$: $x^{(5)} = \{E, F, G\}$ with $w = [1 4 4 3 3 2 1 0 0]$
6. $w(\{F, G, H\}) = w(\{F, G, I\}) = 3 < 9 = w(\{C, E, F\})$: $x^{(6)} = \{F, G, H\}$ with $w = [1 4 4 3 3 3 2 1 0]$

Now the neighborhood $N_{\mathcal{A}_1}$ is not empty: $x^{(7)} = \{F, G, H, I\}$

Example: DLS for the MAX-SAT. Given m logical disjunctions depending on n logical variables, find a truth assignment satisfying the maximum number of clauses

- Neighborhood $N_{\mathcal{F}_1}$ (1-flip) is generated complementing a variable.
- Associate to each logical clause a penalty w_j initially equal to 1 (each component is a satisfied formula).
- The exchange heuristic maximizes the weight of satisfied clauses thus modifying their number with the multiplicative penalty.
- The penalty is updated
 1. increasing the weight of unsatisfied clauses to favor them

$$w_j := \alpha_{us} w_j \text{ for each } j \in U(x) \text{ (with } \alpha_{us} > 1)$$

when a local optimum is reached

2. reducing the penalty towards 1

$$w_j := (1 - \rho)w_j + \rho \cdot 1 \text{ for each } j \in C \text{ (with } \rho \in (0, 1))$$

with a certain probability or after a certain number of updates

The rationale of the method consists in aiming to

- satisfy the currently unsatisfied clauses (diversification)
- in particular, those which have been unsatisfied for longer time and more recently (memory)

The parameters tune intensification and diversification

- small values of α_{us} and ρ preserve the current penalty (intensification)
- large values of α_{us} push away from the current solution (diversification)
- large values of ρ lead push towards the local optimum of the current attraction basin (a different kind of intensification)

6.19 Extending the local search with worsenings

If the neighborhood and objective remain the same, the rule of acceptance must change: instead of

$$x' := \arg \min_{x \in N(x)} f(x)$$

select a nonminimal (possibly, even non-improving) solution.

The main problem is the **risk of cyclically visiting the same solutions.**

The two main strategies that allow to control this risk are

- **Simulated Annealing** (SA), which uses **randomization** to make repetitions unlikely.
- **Tabu Search** (TS), which uses **memory** to forbid repetitions.

6.20 Simulated Annealing

The SA derives from Metropolis' algorithm (1953), which aims to simulate the “annealing” process of metals:

- bring the metal to a temperature close to fusion, so that its particles distribute at random
- cool the metal very slowly, so that the energy decreases, but in a time sufficiently long to converge to thermal equilibrium (all the parts cool down equally, so the particles should diffuse equally)

The aim of the process is to obtain

- a very regular and defectless crystal lattice, that corresponds to the base state (minimum energy configuration)
- a material with useful physical properties

The situation has **similarities with Combinatorial Optimization problems**

- the **states** of the physical system correspond to the **solutions**
- the **energy** corresponds to the **objective function**
- the **base state** corresponds to the **globally optimal solutions** (minima)
- the **state transitions** correspond to **local search moves**
- the **temperature** corresponds to a **numerical parameter**

This suggests to **use Metropolis' algorithm for optimization**.

According to thermodynamics at the thermal equilibrium the probability of observing each state i depends on its energy E_i

$$\pi'_T(i) = \frac{e^{\frac{-E_i}{kT}}}{\sum_{j \in S} e^{\frac{-E_j}{kT}}}$$

where S is the state set, T the temperature and k Boltzmann's constant.

It is a dynamic equilibrium, with ongoing state transitions in all directions.

Metropolis' algorithm generates a **random sequence of states**

- the **current state i has energy E_i**
- the algorithm **perturbs i , generating a state j with energy E_j**
- the **current state moves from i to j with probability**

$$\pi_T(i, j) = \begin{cases} 1 & \text{if } E_j < E_i \\ e^{\frac{E_i - E_j}{kT}} = \frac{\pi'(j)}{\pi'(i)} & \text{if } E_j \geq E_i \end{cases}$$

that is the transition is

- **Deterministic if improving** (because that is the final purpose, if it's lower, and thus better, go to it).
- **Based on the conditional probability if worsening** (ratio of the probability of the final state and the probability of the original state); the probability is proportional to how much worse the state is, it worsens only a little the probability can still be quite big.

The probability of being in the new state j is equal to the product of the probability of being in the old state i times the probability of the transition.

Simulated Annealing applies exactly the same principle.

Algorithm 14 Algorithm *SimulatedAnnealing*($I, x^{(0)}, T^{[0]}$)

```
x := x(0)
x* := x(0)
T := T[0]
while Stop() = false do
    x' := RandomExtract( $N, x$ ) // random uniform extraction
    if  $f(x') < f(x)$  or  $U[0; 1] \leq e^{\frac{f(x)-f(x')}{T}}$  then
        x := x'
    end if
    if  $f(x') < f(x^*)$  then
        x* := x'
    end if
    T := Update(T)
end while
return (x*, f(x*))
```

As the neighborhood is used to generate a solution (not fully explored), **it is possible to worsen even if improving solutions exist.**

A **pre-computed table of values for $e^{\frac{\delta f}{T}}$ can improve the efficiency**; they're real numbers so it can't be fully complete, but it's a random rule so does it really matter if it's approximated?.

Several update schemes can be designed for the “temperature” T .

At each step you extract a random solution from the current neighborhood, if it's better than the old one swap them, if it's worse swap them with a certain probability (given by the extraction of a number between 0 and 1 and checking if it's under the value of the formula).

Check if the solution is better than the best one.

Update T , change the value of the temperature, you're simulating an annealing process in which the temperature decreases.

Accepting a random value from the neighborhood means that there could be a better solution in the neighborhood.

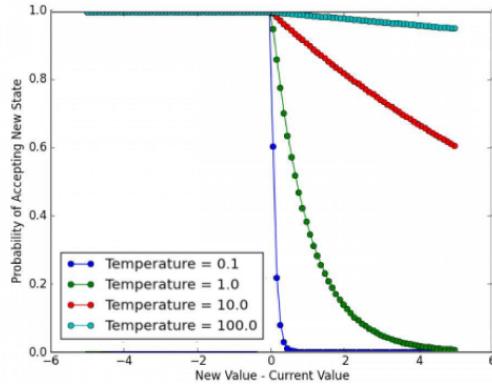
Since there's no exploration, just extracting a random number, the single steps are very fast, but it could require many more steps.

6.20.1 Acceptance criteria

T rules the **probability to accept worsenings**

$$\pi_T(x, x') = \begin{cases} 1 & \text{if } f(x') < f(x) \\ e^{\frac{f(x)-f(x')}{T}} & \text{if } f(x') \geq f(x) \end{cases}$$

- $T \gg 0$ **diversifies** because nearly all solutions are accepted: in the extreme case, it is a random walk
- $T \approx 0$ **intensifies** nearly all worsening solutions are rejected: in the extreme case, it is a steepest descent



Notice the similarity with the ILS.

If you're improving you always accept, if not you accept with a probability based on temperature, the higher the temperature the more the probability to accept the solution increases.

6.20.2 Asymptotic convergence to the optimum

Due to the acceptance rule, **the current solution x is a random variable** (depends on the random seed given): its “**state probability**” $\pi'(x)$ (associates a value of probability to each solution) **combines on all possible predecessors $x^{(t-1)}$**

- the “state probability” $\pi'(x^{(t-1)})$ of the predecessor
- the **probability to choose the move from $x^{(t-1)}$ to x , that is uniform**
- the **probability to accept the move**, that is

$$\pi_T(x^{(t-1)}, x) = \begin{cases} 1 & \text{if } f(x) < f(x^{(t-1)}) \\ e^{\frac{f(x^{(t-1)}) - f(x)}{T}} & \text{if } f(x) \geq f(x^{(t-1)}) \end{cases}$$

The probability of being in a given solution x in step t is the combination of all the possible probabilities of the previous step, combined with the probability of choosing that specific solution from the previous step (uniform probability, it's random) and the probability of accepting said solution (stated in the formula).

As it **depends only on the previous step**, the solution is a **Markov chain**.

For fixed temperature T , the **transition probabilities are stationary**: it is a **homogeneous Markov chain**.

If the search graph for neighborhood N is connected, the **probability to reach each state is > 0** : it is an **irreducible Markov chain**.

Under these assumptions (irreducible and homogeneous Markov chain), **the state probability converges to a stationary distribution independent of the starting state**.

For any starting state, the probability of being in a given solution after a long time tends to assume a stationary distribution, i.e. the behavior of the algorithm is not influenced by the starting solution.

The stationary distribution favors “good” solutions with the same law imposed by thermodynamics on physical systems at thermal equilibrium

$$\pi_T(x) = \frac{e^{-f(x)/T}}{\sum_{x \in X} e^{-f(x)/T}} \text{ for each } x \in X$$

where X is the feasible region and T the “temperature” parameter-

The **distribution converges** to a **limit distribution** as $T \rightarrow 0$

$$\pi(x) = \lim_{T \rightarrow 0} \pi_T(x) = \begin{cases} \frac{1}{|X^*|} & \text{for } x \in X^* \\ 0 & \text{for } x \in X \setminus X^* \end{cases}$$

which corresponds to a **certain convergence to a globally optimal solution**.

This result however holds at the equilibrium, in infinite time.

In practice, **low values of T** imply

- a **high probability to visit a global optimum**, but also
- a **slow convergence** to the optimum (many exchanges are rejected)

In a finite time, the result obtained with **low T** can be **far from optimal**. Hence, **T starts high** and is progressively updated **decreasing over time**.

The **starting value $T^{[0]}$** should be

- **high enough** to allow to **reach any solution quickly**
- **small enough to discourage visiting very bad solutions**

A classical tuning for $T^{[0]}$ is to

- sample the first neighborhood $N(x^{(0)})$
- set a parameter $\beta \in (0, 1)$
- set $T^{[0]}$ to accept on average a fraction β of the sampled solutions

6.20.3 Temperature update

The temperature is **updated by subsequent phases** ($r = 0, \dots, m$)

- **each phase applies a constant value $T^{[r]}$ for $\ell^{[r]}$ iterations**
- **$T^{[r]}$ decreases exponentially from phase to phase**

$$T^{[r]} := \alpha T^{[r-1]} = \alpha^r T^{[0]} \text{ with } \alpha \in (0, 1)$$

- **$\ell^{[r]}$ increases from phase to phase (often linearly) with values related to the diameter of the search graph** (therefore to the size of the instance)

Since T is variable, the Markov chain x is not homogeneous, but

- if T decreases slowly enough, it converges to the global optimum
- good parameters to tune the decrease depend on the instance (namely, on $f(\tilde{x}) - f(x^*)$, where $f(\tilde{x})$ is the second best value of f ; but the best parameter values are not known a priori)

Adaptive SA variants tune the temperature T based on the results

- set T to a value such that a given fraction of $N(x)$ is accepted
- increase T if the solution has not improved for a certain time (diversification); otherwise decrease it (intensification)

6.21 Tabu Search

The Tabu Search (TS) has been proposed by Glover (1986).

It keeps the basic selection rule of steepest descent

$$x' := \arg \min_{x \in N(x)} f(x)$$

without the termination condition (but this implies cycling!).

The TS imposes a tabu to **forbid the solutions already visited**

$$x' := \arg \min_{x \in N(x) \setminus X_V} f(x)$$

where X_V is the set of the already visited solutions.

Exchange heuristics with tabu: An exchange heuristic that explores a neighborhood imposing a tabu on the already visited solutions requires to:

1. **evaluate the feasibility** of each subset produced by the exchanges (unless guaranteed a priori)
2. **evaluate the cost** of each feasible solution
3. **evaluate the tabu status** of each feasible promising solution

in order to **select the feasible best non-tabu solution**.

An elementary way to implement the evaluation of the tabu is

- save the visited solutions in a suitable structure (tabu list)
- check each explored solution making a query on the tabu list

Potential inefficiency of the tabu mechanism: This elementary evaluation of the tabu however is **very inefficient**

- the **comparison of the solutions** at step t requires **time** $O(t)$ (reducible with hash tables or search trees)
- the **number of solutions** visited **grows indefinitely** over time
- the **memory occupation** grows indefinitely over time

The *Cancellation Sequence Method* and the *Reverse Elimination Method* tackle these problems, exploiting the fact that in general

- the **solutions visited** form a **chain with small variations**
- the **solutions visited** are **rarely neighbors** of the **current one**

The idea is to **focus on variations**

- **save move lists**, instead of solutions
- **evaluate the overall performed variations**, instead of the single moves
- **find the solutions which have undergone small overall variations** (recent ones or submitted to variations subsequently reversed)

Potential ineffectiveness of the tabu mechanism: Other subtle phenomena influence the **effectiveness of the method**.

Forbidding the solutions visited can have two different **negative effects**:

- it can **disconnect the search graph**, creating impassable “iron curtains” that block the search (the prohibition should not be permanent)
- it can **slow down the exit from attraction basins**, creating a “gradual filling” effect that slows down the search (the prohibition should be extended)

The two phenomena suggest apparently opposite remedies. How to combine them?

6.21.1 Reducing potential ineffectiveness

Some simple devices can be adopted in order to control these problems.

Attribute-based tabu: Forbid all **solutions that share “attributes” with the visited ones.**

Forbidding only the visited solution slows down the search, so instead

- define a **set A of attributes** (general attributes that any solution can have)
- define **for each solution $x \in X$ a subset of attributes $A_x \subseteq A$** (what attributes does each solution have)
- declare a **subset of tabu attributes $\bar{A} \subseteq A$** (empty at first; what attributes are forbidden)
- **forbid** all the **solutions with tabu attributes**

$$x \text{ is tabu} \Leftrightarrow A_x \cap \bar{A} \neq \emptyset$$

(if the solution has an attribute inside \bar{A} then forbid it)

- **move from** the current solution x **to** x' such that $A_{x'} \cap \bar{A} = \emptyset$ (the new solution has no forbidden attributes) and add to \bar{A} the attributes possessed by x and not by x'

$$\bar{A} := \bar{A} \cup (A_x \setminus A_{x'})$$

(in this way, x becomes tabu)

You don't check a list of tabu solutions, the criteria to forbid a solution is the intersection of its attributes with the set of forbidden ones.

This allows to

- **avoid also solutions similar to the visited ones**
- **get more quickly far away from visited local optima**

The **concept of “attribute”** is intentionally **generic**; the simpler ones are

- **Inclusion of an element in the solution** ($A = B$ and $A_x = x$): when the move from x to x' expels an element i from the solution, the tabu forbids the reinsertion of i in the solution
 - x has the attribute “presence of i ” and x' hasn’t got it
 - the attribute “presence of i ” enters \bar{A}
 - every solution including i becomes tabu
- **Exclusion of an element from the solution** ($A = B$ and $A_x = B \setminus x$): when the move from x to x' inserts an element i into the solution, the tabu forbids the removal of i from the solution
 - x has the attribute “absence of i ” and x' hasn’t got it
 - the attribute “absence of i ” enters \bar{A}
 - every solution devoid of i becomes tabu

Different attribute sets can be combined, each with its tenure and list (e.g., after replacing i with j , forbid both to remove j and to insert i).

Other (less frequent) examples of attributes:

- the **value of the objective function**: forbid solutions of a given value, previously assumed by the objective
- the **value of an auxiliary function**

Complex attributes can be obtained combining simple attributes

- the **coexistence** in the solution **of two elements** (or their separation)
- or, if a move replaces element i with element j , the tabu can **forbid the removal of j to include i** , but allow the simple removal of j and the simple inclusion of i

Other solutions:

Give a limited length L (tabu tenure) to the prohibition: The tabu mechanism creates regions hard or impossible to reach

- the **tabu solutions** become **feasible again after a while**
- the **same solutions can be revisited** (but, if \bar{A} is different, the **future evolution will be different**, if the evolution is the same you're stuck in a cycle)

Tuning the tabu tenure is fundamental for the effectiveness of TS

Introduce an aspiration criteria: The tabu could forbid a global optimum similar to a visited solution, so a tabu solution **better than the best known** one is anyway **accepted** (obviously, there is no risk of looping).

There are looser aspiration criteria, but they are not commonly used.

If all neighbour solutions are tabu, accept the one with the oldest tabu: The tabu could forbid all neighbour solutions. It can be interpreted as another aspiration criteria.

Otherwise the algorithm would just “sit still” and waste time until a tabu expires.

6.21.2 General scheme of the TS

Algorithm 15 Algorithm $\text{TabuSearch}(I, x^{(0)}, L)$

```

 $x := x^{(0)}$ 
 $x^* := x^{(0)}$ 
 $\bar{A} := \emptyset$ 
while  $\text{Stop}() = \text{false}$  do
     $f' := +\infty$ 
    for each  $y \in N(x)$  do
        if  $f(y) < f'$  then
            if  $\text{Tabu}(y, \bar{A}) = \text{false}$  or  $f(y) < f(x^*)$  then
                 $x' := y$ 
                 $f' := f(y)$ 
            end if
        end if
    end for
     $\bar{A} := \text{Update}(\bar{A}, x', L)$ 
    if  $f(x') < f(x^*)$  then
         $x^* := x'$ 
    end if
end while
return  $(x^*, f(x^*))$ 

```

Until the termination condition holds, for each neighbor solution finds which one is the best one; if they're not tabu or if they meet the aspiration criteria, they become the new chosen solution.

After every inner loop it updates the tabu list with the elements of the new solution found and the tenure of already forbidden elements (represented with L).

6.21.3 Efficient evaluation of the tabu status

The **evaluation** of the tabu status must be **efficient** and avoid scanning the whole solution (as for feasibility and cost)

- the **attributes are associated to moves, not to solutions**: do not check whether the solution includes i , but whether the move adds i

Let T_i be the iteration when attribute $i \in A$ became tabu ($-\infty$ if $i \notin \bar{A}$).

To **evaluate** the tabu status in **constant time** simply **check**

$$t \leq T_i + L$$

If the tabu is on insertions ($A = x$), at iteration t

- forbid the moves that add $i \in B \setminus x$ when $t \leq T_i^{in} + L^{in}$
- update $T_i^{in} := t$ for each i removed ($i \in x \setminus x'$)

If the tabu is on deletions ($A = B \setminus x$), at iteration t

- forbid the moves that delete $i \in x$ when $t \leq T_i^{out} + L^{out}$
- update $T_i^{out} := t$ for each i added ($i \in x' \setminus x$)

As either $i \in x$ or $i \in B \setminus x$, a single vector T is enough for both checks (an element can either be in or out).

More sophisticated attributes require more complex structures.

Example: the TSP. Consider the neighborhood $N_{\mathcal{R}_2}$ generated by 2-opt exchanges and use as attributes both the presence and the absence of arcs in the solution

- At first set $T_{ij} = -\infty$ for each arc $(i, j) \in A$.
- At each step t , explore the $n(n-1)/2$ pairs of removable arcs and the corresponding pairs of arcs which would replace them.
- The move (i, j) , which replaces (s_i, s_{i+1}) and (s_j, s_{j+1}) with (s_i, s_j) and (s_{i+1}, s_{j+1}) , is tabu at step t if one of the following conditions holds:
 1. $t \leq T_{s_i, s_{i+1}} + L^{out}$
 2. $t \leq T_{s_j, s_{j+1}} + L^{out}$
 3. $t \leq T_{s_i, s_j} + L^{in}$
 4. $t \leq T_{s_{j+1}, s_{i+1}} + L^{in}$

So, at first all moves are legal. This is basically checking if all the addition and deletion moves respect the tabu conditions; check when the arcs to add were expelled and when the arcs to remove were added, and if it's long enough ago (possibly never) they are not tabu (the value + tenure must be lower than t).

If any of these conditions holds, one of the elements is tabu and the move is forbidden.

- For the selected move (i^*, j^*) , update the auxiliary structures setting
 1. $T_{s_{i^*}, s_{i^*+1}} := t$
 2. $T_{s_{j^*}, s_{j^*+1}} := t$
 3. $T_{s_{i^*}, s_{j^*}} := t$
 4. $T_{s_{j^*+1}, s_{i^*+1}} := t$

Set the value of T to the number of the current iteration for the affected arcs.

As n arcs are in and $n(n-2)$ out of the solution, it is better to set $L^{out} \ll L^{in}$. This also holds for every problem in which the solution is much smaller than the ground set (often).

Example: the Max-SAT. Consider the neighborhood $N_{\mathcal{F}_1}$, which includes the solutions obtained complementing the value of a variable (all n solutions are feasible; one flip neighborhood).

Since $|x| = |B \setminus x|$ for each $x \in X$ (the number of elements in the solution are the same as the number of elements outside)

- the tabu tenure for additions and deletions can be the same
- it is sufficient to forbid the change of value of a variable and the attribute is the variable

The algorithm proceeds as follows

- At first, set $T_i = -\infty$ for each variable $i = 1, \dots, n$.
- at each step t , explore the n solutions obtained complementing each variable.
- the move i , which assigns $x_i := \bar{x}_i$ (flips the variable), is tabu at step t if $t \leq T_i + L$ (it needs to be long enough since the variable has last been flipped; at first all moves are non-tabu).
- perform move i^* and set $T_{i^*} := t$.

6.21.4 Tuning the tabu tenure

The value of the tabu tenure L is a **crucial parameter**

- **too large** tenures can **conceal the global optimum** and in the worst case **block the search**
- **too small** tenures can **hold the exploration back in useless regions** and in the worst case **produce cyclic behaviors**

The **most effective value** of L is in general

- **related to the size of the instance** (big instances require more prohibition)
- **slowly growing with size** (many authors suggest $L \in O(\sqrt{|A|})$, the value of tenure grows with the square root of the value of attributes)
- but nearly **constant on medium ranges of size** (if the size doesn't change very much there's no need to change the tenure)

Cycles can be broken extracting L at **random in a range** $[L_{min}; L_{max}]$.

Adaptive mechanisms update L based on the results of the search within a given range $[L_{min}; L_{max}]$

- **decrease L** when the **current solution x improves**: the search is probably approaching a new local optimum, and we want to favor it (intensification)
- **increase L** when the **current solution x worsens**: the search is probably leaving a known local optimum, and we want to speed up (diversification)

6.21.5 Variants

Other adaptive strategies work in the long term:

- **Reactive Tabu Search:**

- use efficient structures to save the solutions visited (hash table)
- detect cyclic behaviors (frequent repetitions)
- move the range $[L_{min}; L_{max}]$ upwards if the solutions repeat too often

- **Frequency-based Tabu Search:**

- save the frequency of each attribute in the solution in structures similar to the ones used for the tenure (e.g., F_i for each $i \in B$)
- if an attribute appears very often
 - * favor the moves introducing it modifying f as in the DLS
 - * forbid the moves introducing it, or discourage them by modifying f

- **Exploring Tabu Search:** reinitialize the search from solutions of good quality which have been explored, but not used as current solution (i. e., the “second-best solutions” of some neighborhood).

- **Granular Tabu Search:** enlarge or reduce the neighborhood progressively.

7 Recombination Metaheuristics

Constructive and exchange heuristics manage one solution at a time (except for the Ant System).

Recombination heuristics **manage several solutions in parallel**

- **Start from a set** (population) **of solutions** (individuals) obtained somehow.
- **Recombine the individuals** generating a new population.

Their original aspect is the **use of operations working on several solutions**, but they often include features of other approaches (sometimes renamed).

Some are nearly or fully deterministic

- Scatter Search
- Path Relinking

others are strongly randomized (often based on biological metaphors)

- genetic algorithms
- memetic algorithms
- evolution strategies

Of course the effectiveness of a method does not depend on the metaphor.

General scheme: The basic idea is that

- **Good solutions share components** (anything that forms the solution) with the **global optimum**.
- **Different solutions can share different components.**
- **Combining different solutions**, it is possible to **merge optimal components** more easily than building them step by step.

The typical scheme of recombination heuristics is

- Build a **starting population of solutions** (generated in any way).
- As long as a suitable **termination condition does not hold**.
- At **each iteration** (usually called “generation” to distinguish it from more basic concepts of iteration) **update the population**
 - **extract single individuals** and **apply exchange operations** to them
 - **extract subsets of individuals** (usually, pairs) and **apply recombination operations** to them
 - **collect the individuals** thus generated and **choose whether to accept or not** each of them and how many copies **into the new population**

7.1 Scatter Search

Scatter Search (SS), proposed by Glover (1977), proceeds as follows

1. Generate a starting population of solutions.
2. Improve all of them with an exchange procedure.
3. build a **reference set** $R = B \cup D$ where
 - subset B includes the **best known solutions**
 - subset D includes the “**farthest**” **solutions** from B and each other (this requires a distance definition, e.g. the Hamming distance)
4. For each pair of solutions $(x, y) \in B \times (B \cup D)$
 - “recombine” x and y , generating z
 - improve z obtaining z' with an exchange procedure
 - if $z' \notin B$ and B contains a **worse solution**, replace it with z' (we want no duplicates in the reference set); i.e. if it's better than any solution in our set of best solutions replace it.
 - if $z' \notin D$ and D includes a **closer solution**, replace it with z' (we want no duplicates in the reference set); i.e. if it's further away than any solution in our set of farthest solutions replace it.
5. Terminate when R is unchanged.

The rationale is that

- the **recombinations in $B \times B$** intensify the search
- the **recombinations in $B \times D$** diversify the search

Algorithm 16 Algorithm *ScatterSearch*(I, P, n_B, n_D)

```
B := ∅
D := ∅
repeat
    Stop = true
    for each  $x \in P$  do
         $z := \text{SteepestDescent}(I, x)$ 
        if  $f(z) < f(x^*)$  then
             $x^* := z$ 
        end if
         $y_B := \arg \max_{y \in B} f(y)$ 
         $y_D := \arg \min_{y \in D} d(y, B \cup D \setminus \{y\})$ 
        if  $z \notin B$  and ( $|B| < n_B$  or  $f(z) < f(y_B)$ ) then
            //  $B$  keeps the  $n_B$  best unique solutions
             $B := B \cup \{z\}$ 
            Stop := false
        if  $|B| > n_B$  then
             $B := B \setminus \{y_B\}$ 
        end if
    else
        if  $z \notin D$  and ( $|D| < n_D$  or  $d(z, B \cup D \setminus \{y_D\}) > d(y_D, B \cup D \setminus \{y_D\})$ )
        then
            //  $D$  keeps the  $n_D$  most diverse unique solutions
             $D := D \cup \{z\}$ 
            Stop := false
        if  $|D| > n_D$  then
             $D := D \setminus \{y_D\}$ 
        end if
    end if
    end if
    end for
     $P := \emptyset$ 
    for each  $(x, y) \in B \times (B \cup D)$  do
        // Recombine to build the new population
         $P := P \cup \text{Recombine}(x, y, I)$ 
    end for
    until Stop = true
    return  $(x^*, f(x^*))$ 
```

7.1.1 Recombination procedure

The recombination procedure depends on the problem.

Usually, solutions x and y are **manipulated as subsets**

1. Include in z **all the elements shared by x and y :**

$$z := x \cap y$$

both solutions concur in suggesting those elements, it's probably good if both include it.

2. **Augment solution z adding elements from $x \setminus z$ or $y \setminus z$**

- chosen at random or with a greedy selection criteria
- alternatively from each source or freely from the two sources (this is similar to a restricted constructive heuristic)

add elements that are in x or y (and not already in z), it's not choosing elements from the whole ground set.

3. If necessary, **add external elements** from $B \setminus (x \cup y)$.

4. If subset z is **unfeasible**, apply an **auxiliary exchange heuristic** to make it feasible (**repair** procedure).

Examples:

- **MDP**
 - start with $z := x \cap y$
 - augment z with $k - |z|$ random or greedy points from $x \setminus z$ or $y \setminus z$
(add missing points taking them from the solutions)
 - no repair procedure is required
- **Max-SAT**
 - start with $z := x \cap y$
 - augment z with $n - |z|$ random or greedy truth assignments from $x \setminus z$ or $y \setminus z$
 - no repair procedure is required
- **KP**
 - start with $z := x \cap y$
 - augment z with random or greedy elements from $x \setminus z$ or $y \setminus z$ respecting the capacity
 - no repair procedure is required
 - the solution could be augmented with elements from $B \setminus (x \cup y)$
(there could still be space)
- **SCP**
 - start with $z := x \cap y$
 - augment z with random or greedy columns from $x \setminus z$ or $y \setminus z$
(avoiding the redundant ones)
 - remove the redundant columns with a destructive phase

You always start from the intersection, then “fill” the answer with elements from the first two solutions, in the end complete the solution by either augmenting or repairing it.

7.2 Path Relinking

Path Relinking (PR), proposed by Glover (1989), is generally used as a final intensification procedure more than as a stand-alone method.

Given a **neighborhood** N and an **exchange heuristic based on it**

- Collect in a **reference set** R the **best solutions** generated by the **auxiliary heuristic (elite solutions)**.
- **For each pair of solutions** x and y in R
 - build a path γ_{xy} from x to y in the search space of neighborhood N applying to $z^{(0)} = x$ the auxiliary exchange heuristic, but choosing at each step the solution closest to the destination y

$$z^{(k+1)} := \arg \min_{z \in N(z^{(k)})} d(z, y)$$

where d is a suitable metric function on the solutions. In case of equal distance, optimize the objective function f

- find the **best solution** z_{xy}^* along the path (and improve it)

$$z_{xy}^* := \arg \min_{k \in \{1, \dots, |\gamma_{xy}| - 1\}} f(z^{(k)})$$

- if $z_{xy}^* \notin R$ and is **better than the worst in R , add it to R**

Basically, it takes two good solutions from a reference set and finds a (greedy) path between them, searching for an even better solution, which can be added to the reference set, if it's good enough. For every pair the best solution gets added to the population and the process repeated until the algorithm can't improve anymore.

Algorithm 17 Algorithm $\text{PathRelinking}(I, P, n_R)$

```
repeat
     $R := \emptyset$ 
    for each  $x \in P$  do
         $z := \text{SteepestDescent}(I, x)$ 
        if  $f(z) < f(x^*)$  then
             $x^* := z$ 
        end if
         $y_R := \arg \max_{y \in R} f(y)$ 
        if  $z \notin R$  and ( $|R| < n_R$  or  $f(z) < f(y_R)$ ) then
            //  $R$  keeps the  $n_R$  best unique solutions
             $R := R \cup \{z\}$ 
             $Stop := \text{false}$ 
            if  $|R| > n_R$  then
                 $R := R \setminus \{y_R\}$ 
            end if
        end if
    end if
end for
 $P := \emptyset$ 
for each  $x \in R$  and  $y \in R \setminus \{x\}$  do
    // Recombine to build the new population
     $z := x$ 
     $z^* := x$ 
    while  $z \neq y$  do
        // Build a path from  $x$  to  $y$ 
         $Z := \arg \min_{z' \in N(z)} d(z', y)$ 
         $z := \arg \min_{z' \in Z} f(z')$ 
        if  $f(z) < f(z^*)$  then
             $z^* := z$ 
        end if
    end while
    if  $z^* \notin P$  then
         $P := P \cup \{z^*\}$ 
    end if
end for
until  $Stop = \text{true}$ 
return  $(x^*, f(x^*))$ 
```

The paths explored in this way

- **Intensify** the search, because they **connect good solutions**.
- **Diversify** the search, because they **follow different paths** with respect to the exchange heuristic (especially if the extremes are far away)

Since the **distance** of $z^{(k)}$ from y is **decreasing**, one can explore

- **Worsening solutions** without the risk of cyclic behaviors.
- **Unfeasible subsets** without the risk of not getting back to feasibility (they do not improve directly, but open the way to improvements).

7.2.1 Variants

Given two solutions x and y , Path Relinking has several variants:

- **Forward path relinking:** build a path from the worse to the better one.
- **Backward path relinking:** build a path from the better to the worse one.
- **Back-and-forward path relinking:** build both paths.
- **Mixed path relinking:** build a path with alternative steps from each extreme (updating the destination).
- **Truncated path relinking:** build only the first steps of the path (if the good solutions are experimentally close to each other).
- **External path relinking:** build a path from one moving away from the other (if the good solutions are experimentally far from each other).

7.3 Encoding-based algorithms

Many recombination heuristics define and **manipulate encodings of the solutions** (i.e., compact representations), **rather than the solutions**.

The aims of this approach are

- **Abstraction:** conceptually distinguishing the method from the problem to which it is applied.
- **Generality:** build operators effective on every problem represented with a given family of encodings. This doesn't require knowledge of the original problem.

In a strict sense, every representation of a solution in memory is an encoding: the term “encoding” tends to be used for the more involved and compact ones. The difference is blurred.

7.4 Genetic Algorithm

The genetic algorithm, proposed by Holland (1975), is the most famous among the encoding algorithms but there's a large family of similar methods.

It builds and encodes a population $X^{(0)}$, and repeatedly applies:

1. **Selection:** generate a new population starting from the current one
2. **Crossover:** recombine subsets of two or more individuals
3. **Mutation:** modify the individuals (complementary to the crossover)

Algorithm 18 Algorithm *GeneticAlgorithm*($I, X^{(0)}$)

```

 $\Xi^{(0)} := \text{Encode}(X^{(0)})$ 
 $x^* := \arg \min_{x \in X^{(0)}} f(x) \text{ // Best solution found so far}$ 
for  $g = 1$  to  $n_g$  do
     $\Xi := \text{Selection}(\Xi)$ 
     $\Xi := \text{Crossover}(\Xi)$ 
     $x_c := \arg \min_{\xi \in \Xi} f(x(\xi))$ 
    if  $f(x_c) < f(x^*)$  then
         $x^* := x_c$ 
    end if
     $\Xi := \text{Mutation}(\Xi)$ 
     $x_m := \arg \min_{\xi \in \Xi} f(x(\xi))$ 
    if  $f(x_m) < f(x^*)$  then
         $x^* := x_m$ 
    end if
end for
return  $(x^*, f(x^*))$ 

```

Start from a population $X^{(0)}$ (obtained in any way), encode it into $\Xi^{(0)}$, get the best solution, for each of the n_g generation do selection, crossover, check the best, mutation, check the best again. In the end return the best solution found overall.

7.4.1 Features of a good encoding

The performance of a genetic algorithm depends on the encoding.

The **following properties** should be **satisfied** (with decreasing importance)

1. **Each solution should have an encoding**, except for dominated ones; otherwise, there would be unreachable solutions.
2. **Different solutions** should have **different encodings** (or the best solution with a given encoding should be easy to find); otherwise, there would be unreachable solutions.
3. **Each encoding** should correspond to a **feasible solution**; otherwise, the population would include useless individuals. This doesn't necessarily mean that there's a 1 to 1 correspondence, a single solution can have multiple encodings.
4. **Each solution** should correspond to the **same number of encodings**; otherwise, some solutions would be unduly favored.
5. The **encoding and decoding operations** should be **efficient**, otherwise, the algorithm would be inefficient.
6. **Locality: small changes to the encoding** should induce **small changes to the solution**, otherwise intensification and diversification would be impossible.

These conditions **depend very much on the constraints of the problem** (so much for abstraction...).

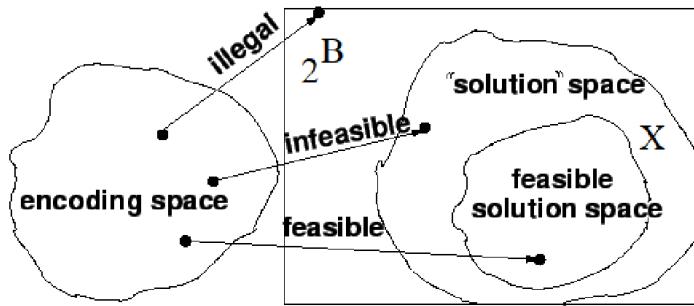
Feasible and unfeasible encodings: Mutation and crossover operators easily produce unfeasible subsets.

If property 3 is not satisfied; this may imply the **violation of**

1. **Quantitative constraints** (e.g., a capacity is exceeded).
2. **Structural constraints** (e.g., the solution is not made of circuits).

The second kind of infeasibility is harder to repair, because it concerns constraints that interact more strongly with each other.

The encoding can lead wherever inside the 2^B possibilities



Often there's a feasible region, but some constraints (usually quantitative) to obtain a “solution space”, with elements that are not solutions, strictly speaking, but they are kinda similar, so we call them unfeasible solutions. E.g., solutions that exceed capacity in the BPP, they will not be feasible but they “look like” a solution, structurally speaking it makes sense.

Some encodings **guarantee structural** (though not quantitative) **feasibility**.

7.4.2 Encodings

The incidence vector: The most direct **encoding** for Combinatorial Optimization problems is the **binary incidence vector** $\xi \in \mathbb{B}^{|B|}$

$$\begin{cases} \xi_i = 1 & \text{indicates that } i \in x \\ \xi_i = 0 & \text{indicates that } i \notin x \end{cases}$$

A generic binary vector **corresponds**

- In the **KP** to a **set of objects**: its weight could be excessive.
- In the **SCP** to a **set of columns**: it could leave uncovered rows.
- In the **PMSP** and in the **BPP** to a **set of assignments of tasks** (objects) **to machines** (containers): it could make zero or more assignments for an element; in the BPP, it could violate the capacity of some containers.
- In the **TSP** to a **set of arcs**: it could not form a Hamiltonian circuit.
- In the **CMSTP (VRP)** to a **set of edges (arcs)**: it could not form a tree (set of cycles), or exceed the capacity of the subtrees (circuits).

Basically, if 0 the element is not in the solution, if 1 it is.

Symbolic strings: If the **ground set** is **partitioned into components** (objects, tasks, Boolean variables, vertices, nodes...)

$$B = \bigcup_{c \in C} B_c \quad \text{with } B_c \cap B_{c'} = \emptyset \text{ for each } c \neq c'$$

and the **feasible solutions contain one element of each component**

$$|x \cap B_c| = 1 \text{ for each } c$$

Example: the assignment of each variable in the Max-SAT problem is a component each, the solution consists of a single value for each of these components (each variable assignment).

One can

- Define for each $c \in C$ an alphabet of symbols **describing component** B_c .
- Encode the solution into a string of symbols $\xi \in B_1 \times \dots \times B_{|C|}$

$$\xi_c = \alpha \text{ indicates that } x \cap B_c = \{(c, \alpha)\}$$

this means that component c is assigned to α .

Define an alphabet for the values of each component and thus the solutions can be described by a sequence of symbols, one for each component. Example, for the BPP by putting in sequence the “names” of the container the first symbol is the assignment of the first object, the second symbol is the assignment of the second object and so on.

Examples of encodings:

- **Max-SAT:** a string of n Boolean values, one for each logical variable.
- **PMSP:** a string of machine labels, one for each task.
- **BPP/CMSTP:** a string of container/subtree labels, one for each object/vertex:
 - the structural constraint on object assignment is enforced
 - the quantitative constraint on capacity is neglected
- For the **VRP**, a string of vehicle labels, one for each node (but capacity is neglected and decoding the circuit for each vehicle is an \mathcal{NP} -complete problem).
- The solutions of the **TSP**, the **KP**, the **SCP** are **not partitions**.

Permutations of a set A common encoding is given by the **permutations of a set**

- If the **solutions are permutations**, this is the **natural encoding** (TSP solutions are subsets of arcs, but also permutations of nodes).
- If the **solutions are partitions** and the **objective is additive** on the subsets, the **order-first split-second method transforms permutations into partitions** (but solutions and encodings do not correspond one-to-one!).
- If the problem **admits a constructive algorithm that at each step**
 1. chooses an element
 2. chooses how to add it to the solution (if many ways exist)we can **feed elements to the algorithm following the permutation** (if step 2 is not unique, some solutions could have no encoding)

7.4.3 Selection

At each generation g a new population $\Xi^{(g)}$ is built extracting $n_p = |\Xi^{(g)}|$ individuals from the current population $\Xi^{(g-1)}$ (typically they have the same cardinality)

$$\Xi^{(g)} := \text{Selection}(\Xi^{(g-1)})$$

The extraction follows two fundamental criteria

1. An individual can be extracted more than once.
2. Better individuals are extracted with higher probability

$$\varphi(\xi) > \varphi(\xi') \implies \pi_\xi \geq \pi_{\xi'}$$

where the **fitness** $\varphi(\xi)$ is a **measure of the quality of individual**
 ξ

- for a maximization problem, commonly

$$\varphi(\xi) = f(x(\xi))$$

(simply the objective function)

- for a minimization problem, commonly

$$\varphi(\xi) = UB - f(x(\xi))$$

where $UB \geq f^*$ is a suitable upper bound on the optimum (this is to have a higher fitness for higher quality of individual)

We need to transform fitness into probability, but how?

Proportional selection The original scheme proposed by Holland (1975) assumed a **probability proportional to fitness**

$$\pi_\xi = \frac{\varphi(\xi)}{\sum_{\xi \in \Xi} \varphi(\xi)}$$

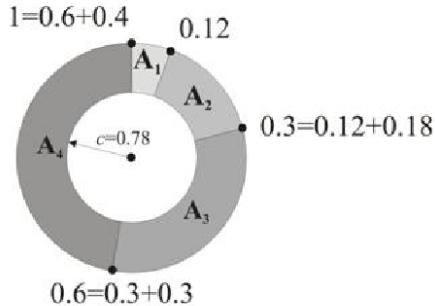
This is named **roulette-wheel selection** or **spinning wheel selection**:

- given the **fitness for** $\xi_i \in \Xi (i = 1, \dots, n_p)$
- **build the intervals**

$$\Gamma_i = \left(\sum_{k=1}^{i-1} \pi_{\xi_k}; \sum_{k=1}^i \pi_{\xi_k} \right]$$

in $O(n_p)$ time

- **extract a random number** $r \in U(0; 1]$
- **choose individual i^* such that** $r \in \Gamma_{i^*}$ in $O(\log n_p)$ time each



You compute all fitness values, normalize it from 0 to 1, assign a segment of $(0; 1]$ to every element. Extract a random value from 0 to 1 which will correspond to a segment and thus an element.

Overall $O(n_p \log n_p)$ time.

The **proportional selection** suffers from

- **stagnation:** in the long term, all individuals tend to have a good fitness, and therefore similar selection probabilities
- **premature convergence:** if the best individuals are bad and the other ones very bad, the selection quickly generates a bad population

Rank selection: To overcome these limitations, one should **at the same time**

- assign **different probabilities to the individuals**
- limit the difference of probability among the individuals

The **rank selection method**

- sorts the individuals by non-decreasing fitness

$$\Xi^{(g)} = \{\xi_1, \dots, \xi_{n_p}\} \text{ with } \varphi_{\xi_1} \leq \dots \leq \varphi_{\xi_{n_p}}$$

- assigns to the k -th individual a **probability equal to**

$$\pi_{\xi_k} = \frac{k}{\sum_{k=1}^{n_p} k} = \frac{2k}{n_p(n_p - 1)}$$

It can be done in $O(n_p \log n_p)$ time as in the previous case.

The probability becomes proportional to the “rank” of an element, the position of each element in the “elements ranking list”. The probability linearly increases with the indices of the sorted sequence.

Tournament selection: An efficient compromise consists in

- Extracting n_p random subsets Ξ_1, \dots, Ξ_{n_p} of size α .
- Selecting the best individual from each subset

$$\xi_k := \arg \max_{\xi \in \Xi_k} \varphi(\xi) \quad k = 1, \dots, n_p$$

In time $O(n_p \alpha)$.

Parameter α tunes the strength of the selection:

- $\alpha \approx n_p$ favors the best individuals
- $\alpha \approx 2$ leaves chances to the bad individuals

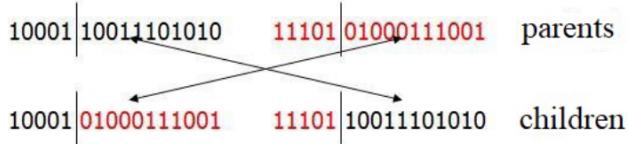
All selection procedures admit an elitist variant, which includes in the new population the best individual of the current one (always keep the best individual found so far).

7.4.4 Crossover

The crossover operator **combines** $k \geq 2$ **individuals** to generate other k . The most common ones set $k = 2$ and are

- **Simple crossover:**

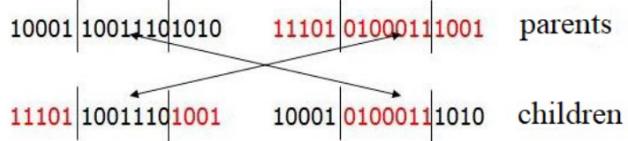
- extract a random position with uniform probability
- split the encoding in two parts at the extracted position
- exchange the final parts of the encodings of the two individuals



Split both encodings in two and swap parts.

- **Double crossover:**

- extract two positions at random with uniform probability
- split the encoding in three parts at the extracted positions
- exchange the extreme parts of the encodings of the two individuals



Same as before but two “split positions”.

Generalizing, one obtains the

- **α points crossover:**

- extract α positions at random with uniform probability
- split the encoding in $\alpha + 1$ parts at the extracted positions
- exchange the odd parts of the encodings of the two individuals (first, third, etc...)

For small values of α , this implies a **positional bias**: symbols **close in the encoding** tend to **remain close**.

To **cancel this bias**, one can adopt the

- **Uniform crossover:**

- build a random binary vector $m \in U(\mathbb{B}^n)$ (“mask”)
- if $m_i = 1$ exchange the symbols in position i of the two individuals, if $m_i = 0$ keep them unmodified

Crossover versus Scatter Search and Path Relinking: The crossover operator resembles the recombination phase of SS and PR.

The main differences are that

1. It **recombines the symbols of the encodings**, instead of
 - recombinig the solutions (SS)
 - performing a chain of exchanges on the solutions (PR)
2. It **operates on the whole population**, instead of only a reference set R (composed of only promising solution).
3. It **operates on random pairs of individuals**, instead of methodically scanning all pairs of solutions in R .
4. It **generates a pair of new individuals**, instead of
 - generating a single intermediate solution (SS)
 - visiting the intermediate solutions and choosing the best one (PR)
5. The **new individuals enter the population directly**, instead of becoming candidates for the reference set.

However, classifying an operator can be a matter of taste.

7.4.5 Mutation

The mutation operator **modifies an individual to generate a similar one**

- scan encoding ξ one symbol at a time
- decide with probability π_m to modify the current symbol

The kind of modification usually depends on the encoding

- **Binary encodings:** flip ξ_i into $\xi'_i := 1 - \xi_i$ (changes value).
- **Symbol strings:** replace ξ_c with a random symbol $\xi'_c \in B_c \setminus \{\xi_c\}$ selected with a uniform probability (change the symbol with another random possible one).
- **Permutations:** there are many proposals
 - exchange two random elements in the permutation (swap)
 - reverse the stretch between two random positions of the permutation
 - ecc...

The symbols must remain all different from each other, otherwise it's not a permutation.

Basically, scan the encoding one symbol at a time and decide, with a certain probability, if mutating it, following one of the modifications described.

The mutation operator has strong relations with exchange operations.

The main differences are that

1. it **modifies the symbols of an encoding**, instead of exchanging elements of a solution
2. it **operates on random symbols**, instead of exploring a neighborhood systematically
3. it **operates on a random subset of symbols of size unknown a priori**, somewhat like sampling a very large scale neighborhood, instead of exchanging a fixed number of elements
4. it **operates on random individuals**, instead of all solutions
5. the **new individuals enter the population directly**, instead of becoming candidates for the reference set

7.4.6 The feasibility problem

If the encoding is not fully invertible, **crossover and mutations sometimes generate encodings that do not correspond to feasible solutions.**

We distinguish between

- **Feasible encodings** that **correspond to feasible solutions.**
- **Unfeasible encodings** that **correspond to legal, but unfeasible subsets.**

The **existence of unfeasible encodings** implies **several disadvantages:**

- **inefficiency:** computational time is lost handling meaningless objects
- **ineffectiveness:** the heuristic explores less solutions (possibly, none)
- **design problems:** fitness must be defined also on unfeasible subsets

There are three main approaches to **face this problem**

1. **Special encodings and operators** (avoid or limit infeasibility).
2. **Repair procedures** (turn infeasibility into feasibility).
3. **Penalty functions** (accept infeasibility, but discourage it).

Special encodings and operators: The idea is to investigate

- **Encodings that (nearly) always yield feasible solutions**, such as
 - permutation encodings and order-first split-second decodings for partition problems (CMSTP, VRP, etc...)
 - permutation encodings and constructive heuristic decodings for scheduling problems (PMSP,...)

For example, if you use different symbols for each element and take an element from each alphabet you guarantee that the solution is a partition (if the problem is a partitioning problem).

- **Crossover and mutation operators that maintain feasibility**, such as
 - operators that simulate moves on solutions (k -exchanges, for example for the TSP)
 - specialized operators (Order or PMX crossover for the TSP)

If you start with a feasible solution, and thus a feasible encoding, if you only simulate moves that are part of the problem and keep the solution in the feasible space, the modified encoding will remain a feasible solution.

These methods

- tend to closely **approximate exchange and recombination heuristics based on the concept of neighborhood**
- **give up the idea of abstraction and focus on the specific problem**, contrary to classical genetic algorithms

So, we're abandoning the idea of "general genetic algorithm" which work on encodings and don't care about the problem (really hard to put in practice, so kinda expected).

Repair procedures: A repair procedure is a **refined decoder function** $x_R : \Xi \rightarrow X$ that

- decodes any encoding ξ into a possibly unfeasible solution $x(\xi) \notin X$
- transforms subset $x(\xi)$ into a feasible solution $x_R(\xi) \in X$
- returns x_R

The procedure is **applied to each unfeasible encoding** $\xi \in \Xi^{(g)}$

- in some methods, **the encoding $\xi(x_R(\xi))$ replaces ξ in $X^{(g)}$** (you don't keep the encoding of the unfeasible solution, you manage only feasible solutions)
- in other ones, **ξ remains in $\Xi^{(g)}$** and $x_R(\xi)$ is **used only to update x^*** (you keep the encoding of the unfeasible solutions, but use only the feasible one to update the best solution)

The methods of the first family

- **Maintain a population of feasible solutions**

but they introduce

- a strong **bias in favor of feasible encodings**
- a **bias in favor of the feasible solutions most easily obtained** with the repair procedure

Penalty functions: Measuring the infeasibility, if the objective function is extended to unfeasible subsets $x \in 2^B \setminus X$, the fitness function $\phi(\xi)$ can be extended to any encoding, but **many unfeasible subsets have a fitness larger than the optimal solution.**

The selection operator tends to **favor such unfeasible subsets** (and the population becomes full of unfeasible subjects).

To avoid that, the **fitness function must combine**

- the **objective value** $f(x(\xi))$
- a **measure of infeasibility** $\psi(x(\xi))$

$$\begin{cases} \psi(x(\xi)) = 0 & \text{if } x(\xi) \in X \\ \psi(x(\xi)) > 0 & \text{if } x(\xi) \notin X \end{cases}$$

Basically, it's zero if it's feasible and it's positive if it's not, proportionally to how infeasible it is. How do we measure infeasibility?

If the constraints of the problem are expressed by equalities or inequalities, $\psi(x)$ can be defined as a weighted sum of their violations.

How to define the weights? Are they fixed, variable or adaptive?.

Definition of the fitness: The most typical combinations are

- **Absolute penalty:** minimize ψ and f lexicographically; given two encodings ξ and ξ' in a rank or tournament selection
 - choose the less unfeasible one
 - if they are equally (un)feasible, choose the better

Feasibility prevails on the cost, if one's feasible and one's not, always choose the first, otherwise choose the better/less unfeasible one.

- **Proportional penalty:** use a linear combination of f and ψ

$$\varphi(\xi) = f(x(\xi)) - \alpha\psi(x(\xi)) + M \text{ for maximization problems}$$

$$\varphi(\xi) = -f(x(\xi)) - \alpha\psi(x(\xi)) + M \text{ for minimization problems}$$

where offset M guarantees that $\varphi(\xi) \geq 0$ for all encodings. The weight α tunes the importance of feasibility.

- **Penalty obtained by repair**, that is keep the unfeasible encoding, but derive its fitness from the objective value of the repaired solution

$$\varphi(\xi) = f(x_R(\xi)) \text{ or } \varphi(\xi) = UB - f(x_R(\xi))$$

since usually $f(x_R(\xi))$ is worse than $f(x(\xi))$ (unfeasible usually better than feasible). The objective function of the repaired solution is the fitness of the original one.

Weight tuning: Experimentally, it is **better** to use the **smallest effective penalty**

- If the penalty is **too small**, **too few feasible solutions are found**.
- If the penalty is **too large**, the **search is confined within a part of the feasible region** (“hidden” feasible solutions are hard to find).

A good value of the parameter α tuning the penalty can be found with

- **Dynamic methods:** increase α over time according to a fixed scheme (first reach good subsets, then enforce feasibility).
- **Adaptive methods:** update α depending on the situation
 - increase α when unfeasible encodings dominate the population (penalize unfeasibility more, go back to feasible solutions)
 - decrease α when feasible encodings dominate (I want better solutions overall, care less about feasibility)
- **Evolutionary methods:** encode α in each individual (not only the current solution), in order to select and refine both the solution and the algorithm parameter. A vector inside the individual keeps the value of α , to encode inside itself how it should move/evolve in the future.

7.5 Memetic algorithms

Memetic algorithms (Moscato, 1989) are inspired by the concept of meme (Dawkins, 1976) that is a basic unit of reproducible cultural information

- genes are selected only at the phenotypic expression level
- memes also adapt directly, as in Lamarckian evolution

Out of the metaphor, **memetic algorithms combine**

- “**Genotypic**” operators that **manipulate the encodings** (crossover and mutation).
- “**Phenotypic**” operators that **manipulate the solutions** (local search).

In short, the **solutions are improved with exchanges before re-encoding**.

Several parameters determine how to apply local search

- how often (at every generation, or after a sufficient diversification)
- to which individuals (all, the best ones, the most diversified ones)
- for how long (until a local optimum, beyond, or stopping before)
- with what method (steepest descent, VNS, ILS, etc...)

The idea is, you take some/all solutions encoded in the populations and improve them with exchanges, you have to decide how to apply exchange heuristics, how to apply the local search.

These are hybridized methods that take into account various other methods already described.

7.6 Evolution strategies

They have been proposed by Rechenberg and Schwefel (1971).

The main differences with respect to classical genetic algorithms are:

- The **solutions** are **encoded** into **real vectors**.
- A **small population of μ individuals** generate **λ candidate descendants** (originally, $\mu = 1$).
- The **new individuals compete to build the new population**
 - in the (μ, λ) strategy the best μ descendants replace the original population, even if some are dominated
 - in the $(\mu + \lambda)$ strategy the best μ individuals overall (predecessors or descendants) survive in the new population
- The **mutation operator sums to the encoding a random noise with a normal distribution of zero average**

$$\xi' := \xi + \delta \text{ with } \delta \in N(0, \sigma)$$

basically a disturbance taken from a distribution.

- Originally, the crossover operator was not used (now it is)

The random-key genetic algorithm (Bean, 1994) use real-vector encodings and decode procedures based on sorting the real numbers.