DM811

Heuristics for Combinatorial Optimization

Lecture 4

Construction Heuristics and Metaheuristics

—

Intro to Experimental Analysis

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

Outline

1) Which variable should we assign next,
and in what order should its values be tried?

- Select-Initial-Unassigned-Variable
- Select-Unassigned-Variable
  - most constrained first = fail-first heuristic
    = Minimum remaining values (MRV) heuristic
    (tend to reduce the branching factor and to speed up pruning)
  - least constrained last

  Eg.: max degree, farthest, earliest due date, etc.
- Order-Domain-Values
  - greedy
  - least constraining value heuristic
    (leaves maximum flexibility for subsequent variable assignments)
  - maximal regret
    implements a kind of look ahead

2) What are the implications of the current variable assignments for the other unassigned variables?

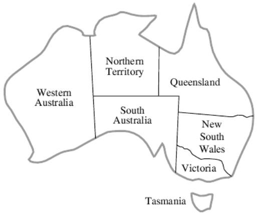
Propagating information through constraints:

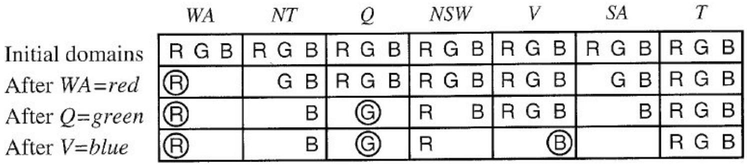
- Implicit in Select-Unassigned-Variable
- Forward checking (coupled with Minimum Remaining Values)
- Constraint propagation in CSP
  - arc consistency: force all (directed) arcs uv to be consistent:
    $\exists$ a value in $D(v)$: $\forall$ values in $D(u)$, otherwise detects inconsistency

    can be applied as preprocessing or as propagation step after each assignment (Maintaining Arc Consistency)

    Applied repeatedly

Propagation: An Example





	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

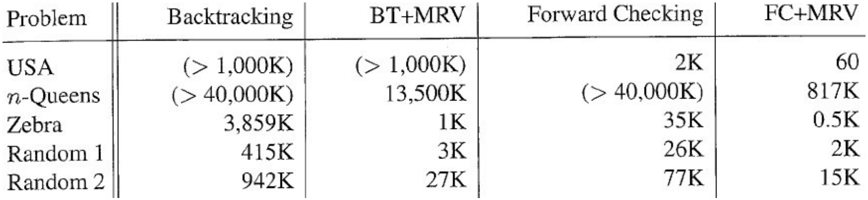
Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

3) When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?

Backtracking-Search

- chronological backtracking, the most recent decision point is revisited
- backjumping, backtracks to the most recent variable in the conflict set (set of previously assigned variables connected to X by constraints).

An Empirical Comparison



Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV
USA	(> 1,000K)	(> 1,000K)	2K	60
n-Queens	(> 40,000K)	13,500K	(> 40,000K)	817K
Zebra	3,859K	1K	35K	0.5K
Random 1	415K	3K	26K	2K
Random 2	942K	27K	77K	15K

Median number of consistency checks

Dealing with Objectives
Optimization Problems

A* search

- The priority assigned to a node x is determined by the function

$$f(x) = g(x) + h(x)$$

  $g(x)$: cost of the path so far
  $h(x)$: heuristic estimate of the minimal cost to reach the goal from x.
- It is optimal if $h(x)$ is an
  - admissible heuristic: *never overestimates* the cost to reach the goal
  - consistent: $h(n) \leq c(n, a, n') + h(n')$

A* best-first search

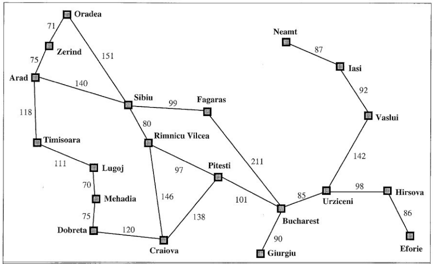


Figure 3.2 A simplified road map of part of Romania.

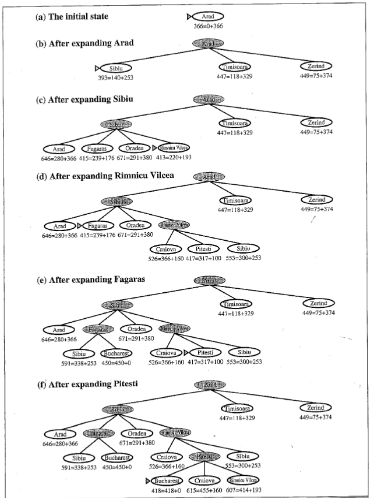


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

A* search

Possible choices for admissible heuristic functions

- optimal solution to an easily solvable **relaxed problem**
- optimal solution to an easily solvable **subproblem**
- learning from experience by gathering statistics on state features
- preferred heuristics functions with higher values (provided they do not overestimate)
- if several heuristics available $h_1, h_2, \ldots, h_m$ and not clear which is the best then:

$$h(x) = \max\{h_1(x), \ldots, h_m(x)\}$$

A* search

Drawbacks

- Time complexity: In the worst case, the number of nodes expanded is exponential,
  (but it is polynomial when the heuristic function h meets the following condition:
  $$|h(x) - h^*(x)| \leq O(\log h^*(x))$$
  h^* is the optimal heuristic, the exact cost of getting from x to the goal.)
- Memory usage: In the worst case, it must remember an exponential number of nodes.
  Several variants: including iterative deepening A* (IDA*),
  memory-bounded A* (MA*) and simplified memory bounded A* (SMA*)
  and recursive best-first search (RBFS)

Outline

Incomplete Search

Complete search is often better suited when ...

- proofs of insolubility or optimality are required;
- time constraints are not critical;
- problem-specific knowledge can be exploited.

Incomplete Search

Complete search is often better suited when ...

- proofs of insolubility or optimality are required;
- time constraints are not critical;
- problem-specific knowledge can be exploited.

Incomplete search is the necessary choice when ...

- non linear constraints and non linear objective function;
- reasonably good solutions are required within a short time;
- problem-specific knowledge is rather limited.

Greedy algorithms

Greedy algorithms (derived from best-first)

- Strategy: always make the choice that is best at the moment
- They are not generally guaranteed to find globally optimal solutions (but sometimes they do: Minimum Spanning Tree, Single Source Shortest Path, etc.)

We will see problem sepcific examples

Outline

Metaheuristics

On backtracking framework
(beyond best-first search)

- Random Restart
- Bounded backtrack
- Credit-based search
- Limited Discrepancy Search
- Barrier Search
- Randomization in Tree Search

Outside the exact framework
(beyond greedy search)

- Random Restart
- Rollout/Pilot Method
- Beam Search
- Iterated Greedy
- GRASP
- (Adaptive Iterated Construction Search)
- (Multilevel Refinement)

Outline

Randomization in Tree Search

The idea comes from complete search: the important decisions are made up in the search tree (backdoors) $\rightsquigarrow$ random selections + restart strategy

Random selections

- randomization in variable ordering:
  - breaking ties at random
  - use heuristic to rank and randmly pick from small factor from the best
  - random pick among heuristics
  - random pick variable with probability depending on heuristic value
- randomization in value ordering:
  - just select random from the domain

Restart strategy

- Example: $S_u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 4, 8, 1, \ldots)$

Outline

Rollout/Pilot Method

Derived from A*

- Each candidate solution is a collection of m components $S = (s_1, s_2, \ldots, s_m)$.
- Master process adds components sequentially to a partial solution $S_k = (s_1, s_2, \ldots s_k)$
- At the k-th iteration the master process evaluates feasible components to add based on an heuristic look-ahead strategy.
- The evaluation function $H(S_{k+1})$ is determined by sub-heuristics that complete the solution starting from S_k
- Sub-heuristics are combined in $H(S_{k+1})$ by
  - weighted sum
  - minimal value

Speed-ups:

- halt whenever cost of current partial solution exceeds current upper bound
- evaluate only a fraction of possible components

Outline

Beam Search

Again based on tree search:

- maintain a set B of bw (beam width) partial candidate solutions
- at each iteration extend each solution from B in fw (filter width) possible ways
- rank each $bw \times fw$ candidate solutions and take the best bw partial solutions
- complete candidate solutions obtained by B are maintained in B_f
- Stop when no partial solution in B is to be extended

Outline

Iterated Greedy

Key idea: use greedy construction

- alternation of construction and deconstruction phases
- an acceptance criterion decides whether the search continues from the new or from the old solution.

Iterated Greedy (IG):

```
determine initial candidate solution $s$
**while** termination criterion is not satisfied **do**
    $r := s$
    (randomly or heuristically) destruct part of $s$
    greedily reconstruct the missing part of $s$
    based on acceptance criterion,
        keep $s$ or revert to $s := r$
```

Extension: Squeaky Wheel

Key idea: solutions can reveal problem structure which maybe worth to exploit.

Use a greedy heuristic repeatedly by prioritizing the elements that create troubles.

Squeaky Wheel

- Constructor: greedy algorithm on a sequence of problem elements.
- Analyzer: assign a penalty to problem elements that contribute to flaws in the current solution.
- Prioritizer: uses the penalties to modify the previous sequence of problem elements. Elements with high penalty are moved toward the front.

Possible to include a local search phase between one iteration and the other

Outline

GRASP
Greedy Randomized Adaptive Search Procedure

Key Idea: Combine randomized constructive search with subsequent local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by local search.

GRASP
Greedy Randomized Adaptive Search Procedure

Key Idea: Combine randomized constructive search with subsequent local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by local search.
- Local search methods often require substantially fewer steps to reach high-quality solutions when initialized using greedy constructive search rather than random picking.

GRASP

Greedy Randomized Adaptive Search Procedure

Key Idea: Combine randomized constructive search with subsequent local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by local search.
- Local search methods often require substantially fewer steps to reach high-quality solutions when initialized using greedy constructive search rather than random picking.
- By iterating cycles of constructive + local search, further performance improvements can be achieved.

Greedy Randomized "Adaptive" Search Procedure (GRASP):

```
while termination criterion is not satisfied do
    generate candidate solution $s$ using
        subsidiary greedy randomized constructive search
```

Greedy Randomized "Adaptive" Search Procedure (GRASP):

```
while termination criterion is not satisfied do
    generate candidate solution $s$ using
        subsidiary greedy randomized constructive search
    perform subsidiary local search on $s$
```

Greedy Randomized "Adaptive" Search Procedure (GRASP):

```
while termination criterion is not satisfied do
    generate candidate solution $s$ using
        subsidiary greedy randomized constructive search
    perform subsidiary local search on $s$
```

- Randomization in *constructive search* ensures that a large number of good starting points for *subsidiary local search* is obtained.

Greedy Randomized "Adaptive" Search Procedure (GRASP):

```
while termination criterion is not satisfied do
    generate candidate solution s using
        subsidiary greedy randomized constructive search
    perform subsidiary local search on s
```

- Randomization in *constructive search* ensures that a large number of good starting points for *subsidiary local search* is obtained.
- Constructive search in GRASP is 'adaptive' (or dynamic): Heuristic value of solution component to be added to a given partial candidate solution may depend on solution components present in it.

Greedy Randomized "Adaptive" Search Procedure (GRASP):

```
while termination criterion is not satisfied do
    generate candidate solution s using
        subsidiary greedy randomized constructive search
    perform subsidiary local search on s
```

- Randomization in *constructive search* ensures that a large number of good starting points for *subsidiary local search* is obtained.
- Constructive search in GRASP is 'adaptive' (or dynamic): Heuristic value of solution component to be added to a given partial candidate solution may depend on solution components present in it.
- Variants of GRASP without local search phase (aka *semi-greedy heuristics*) typically do not reach the performance of GRASP with local search.

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a restricted candidate list (RCL).

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a restricted candidate list (RCL).
- RCLs are constructed in each step using a *heuristic function* h.

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a restricted candidate list (RCL).
- RCLs are constructed in each step using a *heuristic function* h.
  - RCLs based on cardinality restriction comprise the k best-ranked solution components. (k is a parameter of the algorithm.)

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a restricted candidate list (RCL).
- RCLs are constructed in each step using a *heuristic function* h.
  - RCLs based on cardinality restriction comprise the k best-ranked solution components. (k is a parameter of the algorithm.)
  - RCLs based on value restriction comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$,
    where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a restricted candidate list (RCL).
- RCLs are constructed in each step using a *heuristic function* h.
  - RCLs based on cardinality restriction comprise the k best-ranked solution components. (k is a parameter of the algorithm.)
  - RCLs based on value restriction comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$,
    where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)
  - Possible extension: reactive GRASP (*e.g.*, dynamic adaptation of α during search)

Outline

Adaptive Iterated Construction Search

Key Idea: Alternate construction and local search phases as in GRASP, exploiting experience gained during the search process.

Adaptive Iterated Construction Search

Key Idea: Alternate construction and local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.

Adaptive Iterated Construction Search

Key Idea: Alternate construction and local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.
- Initialize all weights to some small value τ_0 at beginning of search process.

Adaptive Iterated Construction Search

Key Idea: Alternate construction and local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.
- Initialize all weights to some small value τ_0 at beginning of search process.
- After every cycle (= constructive + local local search phase), update weights based on solution quality and solution components of current candidate solution.

Adaptive Iterated Construction Search (AICS):
initialise weights

Adaptive Iterated Construction Search (AICS):

```
initialise weights
while termination criterion is not satisfied: do
    generate candidate solution $s$ using
        subsidiary randomized constructive search
```

Adaptive Iterated Construction Search (AICS):

```
initialise weights
while termination criterion is not satisfied: do
    generate candidate solution $s$ using
        subsidiary randomized constructive search
    perform subsidiary local search on $s$
```

Adaptive Iterated Construction Search (AICS):

```
initialise weights
while termination criterion is not satisfied: do
    generate candidate solution $s$ using
        subsidiary randomized constructive search
    perform subsidiary local search on $s$
    adapt weights based on $s$
```

Subsidiary constructive search:

- The solution component to be added in each step of *constructive search* is based on i) *weights* and ii) heuristic function h.

Subsidiary constructive search:

- The solution component to be added in each step of *constructive search* is based on i) *weights* and ii) heuristic function h.
- h can be standard heuristic function as, *e.g.*, used by greedy heuristics

Subsidiary constructive search:

- The solution component to be added in each step of *constructive search* is based on i) *weights* and ii) heuristic function h.
- h can be standard heuristic function as, *e.g.*, used by greedy heuristics
- It is often useful to design solution component selection in constructive search such that any solution component may be chosen (at least with some small probability) irrespective of its weight and heuristic value.

Subsidiary local search:

- As in GRASP, local search phase is typically important for achieving good performance.

Subsidiary local search:

- As in GRASP, local search phase is typically important for achieving good performance.
- Can be based on Iterative Improvement or more advanced LS method (the latter often results in better performance).

Subsidiary local search:

- As in GRASP, local search phase is typically important for achieving good performance.
- Can be based on Iterative Improvement or more advanced LS method (the latter often results in better performance).
- Tradeoff between computation time used in construction phase *vs* local search phase (typically optimized empirically, depends on problem domain).

Weight updating mechanism:

- Typical mechanism: increase weights of all solution components contained in candidate solution obtained from local search.

Weight updating mechanism:

- Typical mechanism: increase weights of all solution components contained in candidate solution obtained from local search.
- Can also use aspects of search history;
  e.g., *current candidate solution* can be used as basis for weight update for additional intensification.

Example: A simple AICS algorithm for the TSP (1/2)

[Based on Ant System for the TSP, Dorigo et al. 1991]

- Search space and solution set as usual (all Hamiltonian cycles in given graph G). However represented in a construction tree T.

Example: A simple AICS algorithm for the TSP (1/2)

[Based on Ant System for the TSP, Dorigo et al. 1991]

- Search space and solution set as usual (all Hamiltonian cycles in given graph G). However represented in a construction tree T.
- Associate weight τ_{ij} with each edge (i, j) in G and T

Example: A simple AICS algorithm for the TSP (1/2)

[Based on Ant System for the TSP, Dorigo et al. 1991]

- Search space and solution set as usual (all Hamiltonian cycles in given graph G). However represented in a construction tree T.
- Associate weight τ_{ij} with each edge (i, j) in G and T
- Use heuristic values $\eta_{ij} := 1/w_{ij}$.

Example: A simple AICS algorithm for the TSP (1/2)

[Based on Ant System for the TSP, Dorigo et al. 1991]

- Search space and solution set as usual (all Hamiltonian cycles in given graph G). However represented in a construction tree T.
- Associate weight τ_{ij} with each edge (i, j) in G and T
- Use heuristic values $\eta_{ij} := 1/w_{ij}$.
- Initialize all weights to a small value τ_0 (parameter).

Example: A simple AICS algorithm for the TSP (1/2)

[Based on Ant System for the TSP, Dorigo et al. 1991]

- Search space and solution set as usual (all Hamiltonian cycles in given graph G). However represented in a construction tree T.
- Associate weight τ_{ij} with each edge (i, j) in G and T
- Use heuristic values $\eta_{ij} := 1/w_{ij}$.
- Initialize all weights to a small value τ_0 (parameter).
- *Constructive search* start with randomly chosen vertex
  and iteratively extend partial round trip ϕ by selecting vertex
  not contained in ϕ with probability

$$\frac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in N'(i)} [\tau_{il}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}$$

Example: A simple AICS algorithm for the TSP (2/2)

- *Subsidiary local search* = typical iterative improvement

Example: A simple AICS algorithm for the TSP (2/2)

- *Subsidiary local search* = typical iterative improvement
- *Weight update* according to

$$\tau_{ij} := (1-\rho)\cdot\tau_{ij} + \Delta(ij, s')$$

where $\Delta(i,j,s') := 1/f(s')$, if edge ij is contained in the cycle represented by s', and 0 otherwise.

Example: A simple AICS algorithm for the TSP (2/2)

- *Subsidiary local search* = typical iterative improvement
- *Weight update* according to

  $$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(ij, s')$$

  where $\Delta(i, j, s') := 1/f(s')$, if edge ij is contained in the cycle represented by s', and 0 otherwise.
- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.

Example: A simple AICS algorithm for the TSP (2/2)

- *Subsidiary local search* = typical iterative improvement
- *Weight update* according to

  $$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(ij, s')$$

  where $\Delta(i, j, s') := 1/f(s')$, if edge ij is contained in the cycle represented by s', and 0 otherwise.
- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.
- Decay mechanism (controlled by parameter ρ) helps to avoid unlimited growth of weights and lets algorithm forget past experience reflected in weights.

Example: A simple AICS algorithm for the TSP (2/2)

- *Subsidiary local search* = typical iterative improvement
- *Weight update* according to

  $$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(ij, s')$$

  where $\Delta(i, j, s') := 1/f(s')$, if edge ij is contained in the cycle represented by s', and 0 otherwise.
- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.
- Decay mechanism (controlled by parameter ρ) helps to avoid unlimited growth of weights and lets algorithm forget past experience reflected in weights.
- (Just add a population of cand. solutions and you have an Ant Colony Optimization Algorithm!)

Outline

Fairness Principle

Fairness principle: being completely fair is perhaps impossible but try to remove any possible bias

- possibly all algorithms must be implemented with the same style, with the same language and sharing common subprocedures and data structures
- the code must be optimized, e.g., using the best possible data structures
- running times must be comparable, e.g., by running experiments on the same computational environment (or redistributing them randomly)

Outline

Definitions

The most typical scenario considered in analysis of search heuristics

Asymptotic heuristics with time (or iteration) limit decided *a priori*

The algorithm $\mathcal{A}^\infty$ is halted when time expires.

Deterministic case: $\mathcal{A}^\infty$ on π returns a solution of cost x.

The performance of $\mathcal{A}^\infty$ on π is a scalar $y = x$.

Randomized case: $\mathcal{A}^\infty$ on π returns a solution of cost X, where X is a random variable.

The performance of $\mathcal{A}^\infty$ on π is the univariate $Y = X$.

[This is not the only relevant scenario: to be refined later]

Random Variables and Probability

Statistics deals with random (or stochastic) variables.

A variable is called random if, prior to observation, its outcome cannot be predicted with certainty.

The uncertainty is described by a probability distribution.

Random Variables and Probability

Statistics deals with random (or stochastic) variables.

A variable is called random if, prior to observation, its outcome cannot be predicted with certainty.

The uncertainty is described by a probability distribution.

Discrete variables

Probability distribution:

$$p_i = P[x = v_i]$$

Cumulative Distribution Function (CDF)

$$F(v) = P[x \leq v] = \sum_i p_i$$

Mean

$$\mu = E[X] = \sum x_i p_i$$

Variance

$$\sigma^2 = E[(X-\mu)^2] = \sum (x_i - \mu)^2 p_i$$

Continuous variables

Probability density function (pdf):

$$f(v) = \frac{dF(v)}{dv}$$

Cumulative Distribution Function (CDF):

$$F(v) = \int_{-\infty}^{v} f(v) dv$$

Mean

$$\mu = E[X] = \int x f(x) dx$$

Variance

$$\sigma^2 = E[(X-\mu)^2] = \int (x-\mu)^2 f(x)\, dx$$

Generalization

For each general problem Π (e.g., TSP, GCP) we denote by C_Π a set (or class) of instances and by $\pi \in C_\Pi$ a single instance.

Generalization

For each general problem Π (e.g., TSP, GCP) we denote by C_Π a set (or class) of instances and by $\pi \in C_\Pi$ a single instance.

On a specific instance, the random variable Y that defines the performance measure of an algorithm is described by its probability distribution/density function

$$Pr(Y = y \mid \pi)$$

Generalization

For each general problem Π (e.g., TSP, GCP) we denote by C_Π a set (or class) of instances and by $\pi \in C_\Pi$ a single instance.

On a specific instance, the random variable Y that defines the performance measure of an algorithm is described by its probability distribution/density function

$$Pr(Y = y \mid \pi)$$

It is often more interesting to generalize the performance on a class of instances C_Π, that is,

$$Pr(Y = y, C_\Pi) = \sum_{\pi \in \Pi} Pr(Y = y \mid \pi) Pr(\pi)$$

Sampling

In experiments,

1. we sample the population of instances and
2. we sample the performance of the algorithm on each sampled instance

If on an instance π we run the algorithm r times then we have r replicates of the performance measure Y, denoted $Y_1, \ldots, Y_r$, which are independent and identically distributed (i.i.d.), i.e.

$$Pr(y_1, \ldots, y_r | \pi) = \prod_{j=1}^{r} Pr(y_j \mid \pi)$$

$$Pr(y_1, \ldots, y_r) = \sum_{\pi \in C_\Pi} Pr(y_1, \ldots, y_r \mid \pi) Pr(\pi).$$

Instance Selection

In real-life applications a simulation of $p(\pi)$ can be obtained by historical data.

Instance Selection

In real-life applications a simulation of $p(\pi)$ can be obtained by historical data.

In simulation studies instances may be:

- real world instances
- random variants of real world-instances
- online libraries
- randomly generated instances

Instance Selection

In real-life applications a simulation of $p(\pi)$ can be obtained by historical data.

In simulation studies instances may be:

- real world instances
- random variants of real world-instances
- online libraries
- randomly generated instances

They may be grouped in classes according to some features whose impact may be worth studying:

- type (for features that might impact performance)
- size (for scaling studies)
- hardness (focus on hard instances)
- application (e.g., CSP encodings of scheduling problems), ...

Instance Selection

In real-life applications a simulation of $p(\pi)$ can be obtained by historical data.

In simulation studies instances may be:

- real world instances
- random variants of real world-instances
- online libraries
- randomly generated instances

They may be grouped in classes according to some features whose impact may be worth studying:

- type (for features that might impact performance)
- size (for scaling studies)
- hardness (focus on hard instances)
- application (e.g., CSP encodings of scheduling problems), ...

Within the class, instances are drawn with uniform probability $p(\pi) = c$

Statistical Methods

The analysis of performance is based on finite-sized sampled data.
Statistics provides the methods and the mathematical basis to

- describe, summarizing, the data (descriptive statistics)
- make inference on those data (inferential statistics)

Statistical Methods

The analysis of performance is based on finite-sized sampled data.
Statistics provides the methods and the mathematical basis to

- describe, summarizing, the data (descriptive statistics)
- make inference on those data (inferential statistics)

Statistics helps to

- guarantee reproducibility
- make results reliable
  (are the observed results enough to justify the claims?)
- extract relevant results from large amount of data

Statistical Methods

The analysis of performance is based on finite-sized sampled data.
Statistics provides the methods and the mathematical basis to

- describe, summarizing, the data (descriptive statistics)
- make inference on those data (inferential statistics)

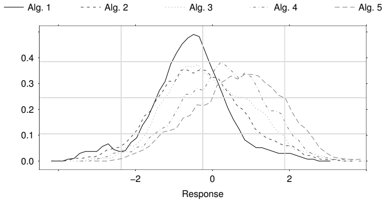
Statistics helps to

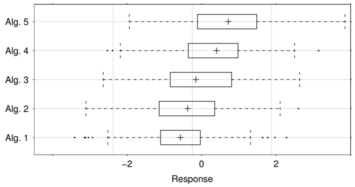
- guarantee reproducibility
- make results reliable
  (are the observed results enough to justify the claims?)
- extract relevant results from large amount of data

In the practical context of heuristic design and implementation (i.e., engineering), statistics helps to take correct design decisions with the least amount of experimentation

Objectives of the Experiments

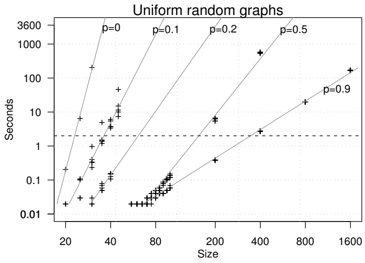
- **Comparison**:
  bigger/smaller, same/different,
  Algorithm Configuration,
  Component-Based Analysis
  - Standard statistical methods:
    experimental designs, test hypothesis and estimation





Objectives of the Experiments

- **Comparison**:
  bigger/smaller, same/different,
  Algorithm Configuration,
  Component-Based Analysis
  - Standard statistical methods: *experimental designs, test hypothesis and estimation*
- **Characterization**:
  Interpolation: fitting models to data
  Extrapolation: building models of data, explaining phenomena
  - Standard statistical methods: *linear and non linear regression* model fitting



Outline

Measures and Transformations

On a single instance

Computational effort indicators

- number of elementary operations/algorithmic iterations (e.g., search steps, objective function evaluations, number of visited nodes in the search tree, consistency checks, etc.)
- total CPU time consumed by the process (sum of *user* and *system* times returned by `getrusage`)

Measures and Transformations

On a single instance

Computational effort indicators

- number of elementary operations/algorithmic iterations (e.g., search steps, objective function evaluations, number of visited nodes in the search tree, consistency checks, etc.)
- total CPU time consumed by the process (sum of *user* and *system* times returned by `getrusage`)

Solution quality indicators

- value returned by the cost function
- error from optimum/reference value
- (optimality) gap $\frac{|UB-LB|}{UB}$
- ranks

Measures and Transformations

On a class of instances

Computational effort indicators

- no transformation if the interest is in studying scaling
- standardization if a fixed time limit is used
- geometric mean (used for a set of numbers whose values are meant to be multiplied together or are exponential in nature),
- otherwise, better to group homogeneously the instances

Solution quality indicators

Different instances imply different scales $\Rightarrow$ need for an invariant measure

(However, many other measures can be taken both on the algorithms and on the instances [McGeoch, 1996])

Measures and Transformations

On a class of instances (cont.)

Solution quality indicators

- Distance or error from a reference value (assume minimization case):

$$e_1(x,\pi) = \frac{x(\pi) - \bar{x}(\pi)}{\sqrt{\hat{\sigma(\pi)}}} \qquad \text{standard score}$$

$$e_2(x,\pi) = \frac{x(\pi) - x^{opt}(\pi)}{x^{opt}(\pi)} \qquad \text{relative error}$$

$$e_3(x,\pi) = \frac{x(\pi) - x^{opt}(\pi)}{x^{worst}(\pi) - x^{opt}(\pi)} \qquad \text{invariant [Zemel, 1981]}$$

  - optimal value computed exactly or known by construction
  - surrogate value such bounds or best known values
- Rank (no need for standardization but loss of information)

Outline

Summary Measures

Measures to describe or characterize a population

- Measure of central tendency, location
- Measure of dispersion

One such a quantity is

- a **parameter** if it refers to the population (Greek letters)
- a **statistics** if it is an *estimation* of a population parameter from the sample (Latin letters)

Measures of central tendency

- Arithmetic Average (Sample mean)

$$\bar{X} = \frac{\sum x_i}{n}$$

- *Quantile*: value above or below which lie a fractional part of the data (used in nonparametric statistics)
  - Median

    $$\mathcal{M} = x_{(n+1)/2}$$

  - Quartile

    $$Q_1 = x_{(n+1)/4} \qquad Q_3 = x_{3(n+1)/4}$$

  - q-quantile

    q of data lies below and $1 - q$ lies above
- Mode

  value of relatively great concentration of data
  (*Unimodal* vs *Multimodal* distributions)

Measure of dispersion

- Sample range

$$R = x_{(n)} - x_{(1)}$$

- Sample variance

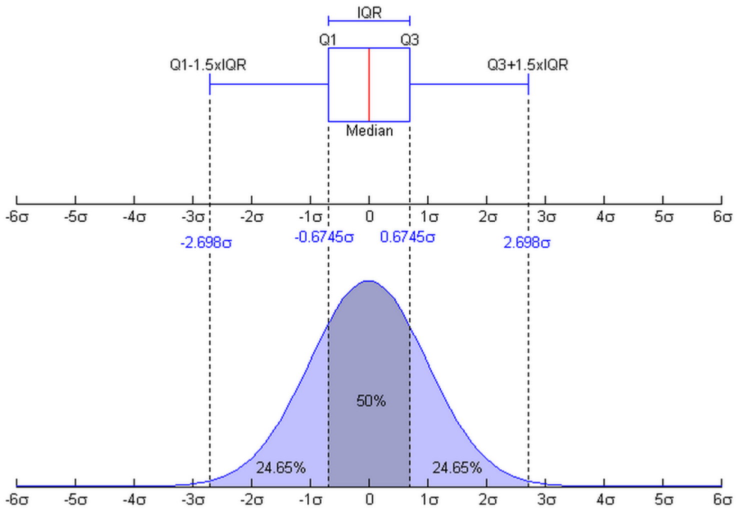
$$s^2 = \frac{1}{n-1} \sum (x_i - \bar{X})^2$$

- Standard deviation

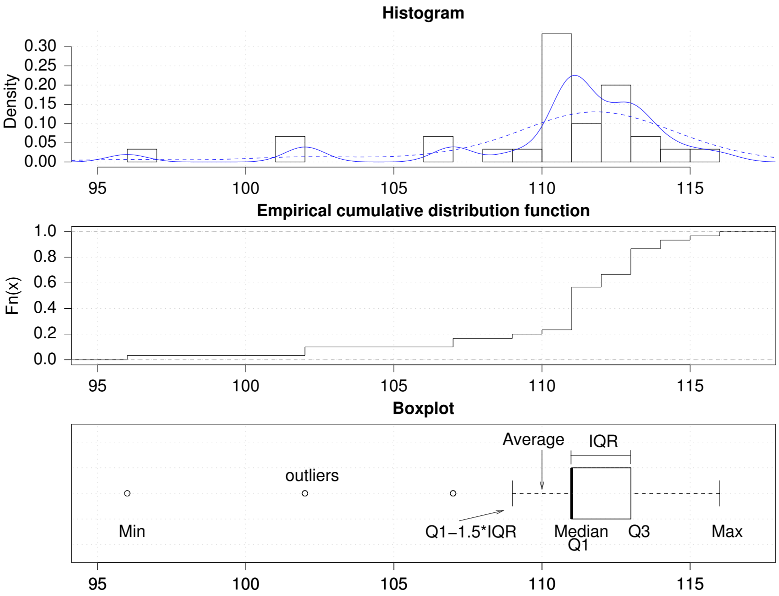
$$s = \sqrt{s^2}$$

- Inter-quartile range

$$IQR = Q_3 - Q_1$$



Boxplot and a probability density function (pdf) of a Normal N(0,1s2) Population. (source: Wikipedia)
[see also: `http://informationandvisualization.de/blog/box-plot`]

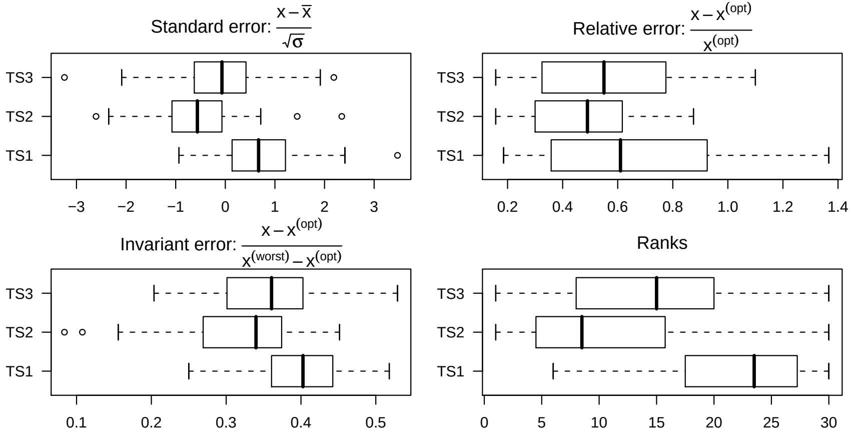

Histogram
Density
0.30
0.25
0.20
0.15
0.10
0.05
0.00
95
100
105
110
115
Empirical cumulative distribution function
Fn(x)
1.0
0.8
0.6
0.4
0.2
0.0
95
100
105
110
115
Boxplot
Average
IQR
outliers
Min
Q1−1.5*IQR
Median
Q1
Q3
Max
95
100
105
110
115

In R

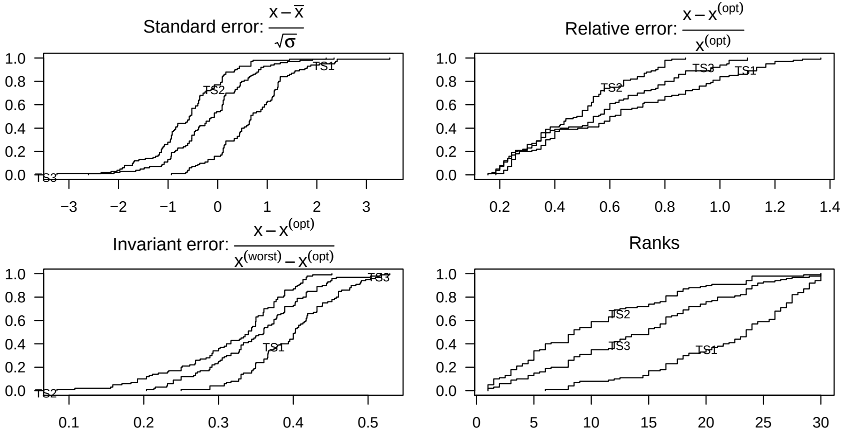
```
> x<-runif(10,0,1)
 mean(x), median(x), quantile(x), quantile(x,0.25)
 range(x), var(x), sd(x), IQR(x)
> fivenum(x)
#(minimum, lower-hinge, median, upper-hinge, maximum)
[1] 0.18672 0.26682 0.28927 0.69359 0.92343
> summary(x)
> aggregate(x,list(factors),median)
> boxplot(x)
```

Outline

On a class of instances



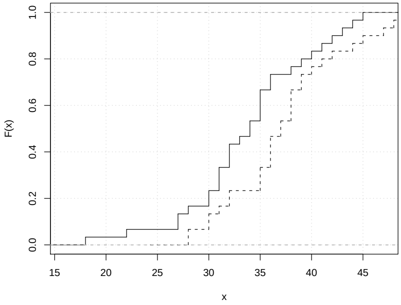
On a class of instances

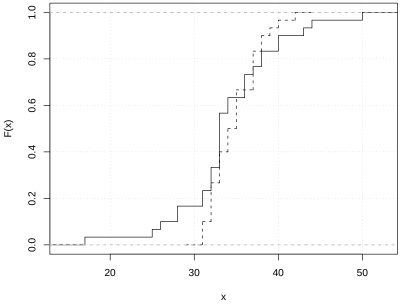


Stochastic Dominance

Definition: Algorithm $\mathcal{A}_1$ probabilistically dominates algorithm $\mathcal{A}_2$ on a problem instance, iff its CDF is always "below" that of $\mathcal{A}_2$, *i.e.*:

$$F_1(x) \leq F_2(x), \qquad \forall x \in X$$





R code behind the previous plots

We load the data and plot the comparative boxplot for each instance.

```
> load("TS.class-G.dataR")
> G[1:5,]
  alg inst run sol time.last.imp tot.iter parz.iter exit.iter exit.time opt
1 TS1 G-1000-0.5-30-1.1.col 1 59 9.900619 5955 442 5955 10.02463 30
2 TS1 G-1000-0.5-30-1.1.col 2 64 9.736608 3880 130 3958 10.00062 30
3 TS1 G-1000-0.5-30-1.1.col 3 64 9.908618 4877 49 4877 10.03263 30
4 TS1 G-1000-0.5-30-1.1.col 4 68 9.948622 6996 409 6996 10.07663 30
5 TS1 G-1000-0.5-30-1.1.col 5 63 9.912620 3986 52 3986 10.04063 30
>
> library(lattice)
> bwplot(alg ~ sol | inst,data=G)
```

R code behind the previous plots

We load the data and plot the comparative boxplot for each instance.

```
> load("TS.class-G.dataR")
> G[1:5,]
  alg inst run sol time.last.imp tot.iter parz.iter exit.iter exit.time opt
1 TS1 G-1000-0.5-30-1.1.col 1 59 9.900619 5955 442 5955 10.02463 30
2 TS1 G-1000-0.5-30-1.1.col 2 64 9.736608 3880 130 3958 10.00062 30
3 TS1 G-1000-0.5-30-1.1.col 3 64 9.908618 4877 49 4877 10.03263 30
4 TS1 G-1000-0.5-30-1.1.col 4 68 9.948622 6996 409 6996 10.07663 30
5 TS1 G-1000-0.5-30-1.1.col 5 63 9.912620 3986 52 3986 10.04063 30
>
> library(lattice)
> bwplot(alg ~ sol | inst,data=G)
```

If we want to make an aggregate analysis we have the following choices:

- maintain the raw data,
- transform data in standard error,
- transform the data in relative error,
- transform the data in an invariant error,
- transform the data in ranks.

Maintain the raw data

```
> par(mfrow=c(3,2),las=1,font.main=1,mar=c(2,3,3,1))
> #original data
> boxplot(sol~alg,data=G,horizontal=TRUE,main="Original data")
```

Transform data in standard error

```
> #standard error
> T1 <- split(G$sol,list(G$inst))
> T2 <- lapply(T1,scale,center=TRUE,scale=TRUE)
> T3 <- unsplit(T2,list(G$inst))
> T4 <- split(T3,list(G$alg))
> T5 <- stack(T4)
> boxplot(values~ind,data=T5,horizontal=TRUE,main=expression(paste("
    Standard error: ",frac(x-bar(x),sqrt(sigma)))))
> library(latticeExtra)
> ecdfplot(~values,group=ind,data=T5,main=expression(paste("Standard error:
",frac(x-bar(x),sqrt(sigma)))))

> #standard error
> G$scale <- 0
> split(G$scale, G$inst) <- lapply(split(G$sol, G$inst), scale,center=TRUE,
    scale=TRUE)
```

Transform the data in relative error

```
> #relative error
> G$err2 <- (G$sol-G$opt)/G$opt
> boxplot(err2~alg,data=G,horizontal=TRUE,main=expression(paste("Relative
    error: ",frac(x-x^(opt),x^(opt)))))
> ecdfplot(G$err2,group=G$alg,main=expression(paste("Relative error: ",frac
    (x-x^(opt),x^(opt)))))
```

Transform the data in an invariant error
We use as surrogate of x^{worst} the median solution returned by the simplest algorithm for the graph coloring, that is, the ROS heuristic.

```
> #error 3
> load("ROS.class-G.dataR")
> F1 <- aggregate(F$sol,list(inst=F$inst),median)
> F2 <- split(F1$x,list(F1$inst))
> G$ref <- sapply(G$inst,function(x) F2[[x]])
> G$err3 <- (G$sol-G$opt)/(G$ref-G$opt)
> boxplot(err3~alg,data=G,horizontal=TRUE,main=expression(paste("Invariant
    error: ",frac(x-x^(opt),x^(worst)-x^(opt)))))
> ecdfplot(G$err3,group=G$alg,main=expression(paste("Invariant error: ",
    frac(x-x^(opt),x^(worst)-x^(opt)))))
```

Transform the data in ranks

```
> #rank
> G$rank <- G$sol
> split(G$rank, G$inst) <- lapply(split(G$sol, D$inst), rank)
> bwplot(rank~reorder(alg,rank,median),data=G,horizontal=TRUE,main="Ranks")
> ecdfplot(rank,group=alg,data=G,main="Ranks")
```