CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Metaheuristic Optimization

Local Search

Dr. Diarmuid Grimes

RTD material based on:
Hoos, Holger H., and Stützle, Thomas. *Stochastic local search: Foundations and applications*. Elsevier, 2004
https://iridia.ulb.ac.be/~stuetzle/Teaching/HO15/Slides/ch4-slides.pdf

Classical search vs. Local Search

Classical search	Local Search
➢ systematic exploration of search space. ➢ Keeps one or more paths in memory. ➢ Records which alternatives have been explored at each point along the path. ➢ The **path** to the goal is a **solution** to the problem.	➢ In many optimization problems, the path to the goal is **irrelevant; the goal state itself is the solution**. ➢ State space = set of "complete" configurations. ➢ Find configuration satisfying constraints, Find best state according to some objective function h(s). e.g., **n-queens**, h(s)= number of attacking queens. **In such cases, we can use Local Search Algorithms.**

Search Process

- **Intensification**:
  - Searches solutions similar to the current solution
  - Want to intensively explore known promising areas of the search space
  - Create solutions using the more attractive components of the best solutions in memory
  - Another technique consists in changing the neighborhood's structure by allowing different moves
- **Diversification**:
  - Examines unvisited regions of the search space
  - Generates different solutions
    - Using components rarely present in the current solution
    - Biasing the evaluation of a move by modifying the objective function adding a term related to component frequencies
- Intensification and diversification phases alternate during the search


POOR
FAIR
GOOD
GREAT


PERFORMANCE
EVALUATION

Las Vegas

Monte Carlo

Gambles
with runtime

Monte Carlo Algorithms

Gambles
with correctness

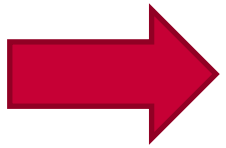
Las Vegas Algorithms

Types of Randomized Algorithms

Randomized Las Vegas Algorithms:

- Output is always correct
- Running time is a **random variable**

Example: Randomized Quick Sort

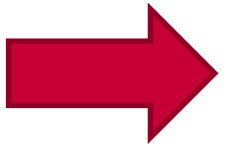


Gambles with runtime

Randomized Monte Carlo Algorithms:

- Output may be incorrect with some probability
- Running time is deterministic.

Example: Primality Tests (e.g. Miller-Rabin)



Gambles with correctness

Las Vegas Algorithms

- Stochastic Local Search (**SLS**) algorithms are typically **incomplete**: there is no guarantee that an (optimal) solution for a given problem instance will eventually be found
- **But**: for decision problems, any solution returned is guarantee to be correct
- Also: The runtime required for finding a solution (in case one is found) is subject to random variation

These properties define the class of (generalized) Las Vegas algorithms, of which SLS algorithms are a subset

Empirical Analysis of Local Search

Analysis of Randomized Algorithms

- How long does it take to find a feasible solution?
- How good are the solutions generated by the algorithm?
- How robust is the heuristic algorithm w.r.t different instances?
- How robust is the heuristic algorithm w.r.t. different parameter settings?
- How does the algorithm scale to large instance size?
- Which is the best of these ***x*** different neighborhood structures?
- Which is the best of these ***x*** different heuristic algorithms?

Analysis of Randomized Algorithms

Ideally: do theoretical analysis, but:

- The theoretical analysis of detailed behavior of (advanced) randomized algorithms is often very difficult due to:
  - Stochasticity of algorithms
  - Complexity of problems (NP-hard)
  - Many degrees of freedom in algorithms which make them theoretically hard to capture
- Practical applicability of theoretical results often limited
  - Rely on assumptions that do not apply in practical situations (e.g., convergence results for Simulated Annealing)
  - Capture only asymptotic behavior and do not reflect actual behavior with sufficient accuracy
  - Apply to worst-case or highly idealized average-case

Analysis of Randomized Algorithms

- Analyze the behavior of randomized algorithms using **sound empirical** methodologies
- Example: follow a scientific procedure:
  - Make observations
  - Formulate hypothesis/hypotheses
  - While not satisfied with model (and deadline not exceeded)
    - Design **computational experiment** to test model
    - Conduct computational experiment
    - Analyse experimental results
    - Revise model based on results

Goals of empirical analysis

Obtain insights into algorithmic performance

- Help assessing suitability for applications
- Provide basis for comparing algorithms
- Characterize algorithm behavior
- Facilitate improvements of algorithms

Goals of empirical analysis

Deconstructing Nowicki and Smutnicki's *i*-TSAB tabu search algorithm for the job-shop scheduling problem☆,☆☆

Jean-Paul Watson[a,*], Adele E. Howe[b], L. Darrell Whitley[b]

[a]*Sandia National Laboratories, P.O. Box 5800, MS 1110, Albuquerque, NM 87185-1110, USA*
[b]*Department of Computer Science, Colorado State University, Fort Collins, CO 80523-1873, USA*

Available online 10 October 2005

Abstract

Over the last decade and a half, tabu search algorithms for machine scheduling have gained a near-mythical reputation by consistently equaling or establishing state-of-the-art performance levels on a range of academic and real-world problems. Yet, despite these successes, remarkably little research has been devoted to developing an understanding of why tabu search is so effective on this problem class. In this paper, we report results that provide significant progress in this direction. We consider Nowicki and Smutnicki's *i*-TSAB tabu search algorithm, which represents the current state-of-the-art for the makespan-minimization form of the classical job-shop scheduling problem. Via a series of controlled experiments, we identify those components of *i*-TSAB that enable it to achieve state-of-the-art performance levels. In doing so, we expose a number of misconceptions regarding the behavior and/or benefits of tabu search and other local search metaheuristics for the job-shop problem. Our results also serve to focus future research, by identifying those specific directions that are most likely to yield further improvements in performance.

https://doi.org/10.1016/j.cor.2005.07.016

Empirical Evaluation – Traditional Approach

Run LS algorithm n times with time limit t_{max}

- Compute:
  - Average
  - Median
  - Standard deviations
  - ?
  - Min, max, $1^{st}/3^{rd}$ quartiles, etc

Purpose: Provide good understanding of typical behavior of algorithm for other users who may wish to consider it.

E.g. Non-experts solving similar instances:

- If I've 100 machines I'm okay if it has large variation and finds really good solutions
- If I've 2 machines I want one that consistently found decent solutions

Examine aspects of stochastic search performance

- Variability due to randomization
- Robustness w.r.t. parameter settings
- Robustness across different instances / instance types
- Scaling with instance size

Run-time Distributions of Randomized Algorithms

- More detailed point of view
  - For decision SLS algorithms the runtime to reach a solution is a random variable
    - (univariate) run-time distributions (RTDs)
  - For optimization LS algorithms the runtime and the solution quality returned are random variables
    - (bivariate) run-time distributions (RTDs)

- Study distribution of random variables characterizing run-time and solution quality of algorithm on given problem instance

Random Variables

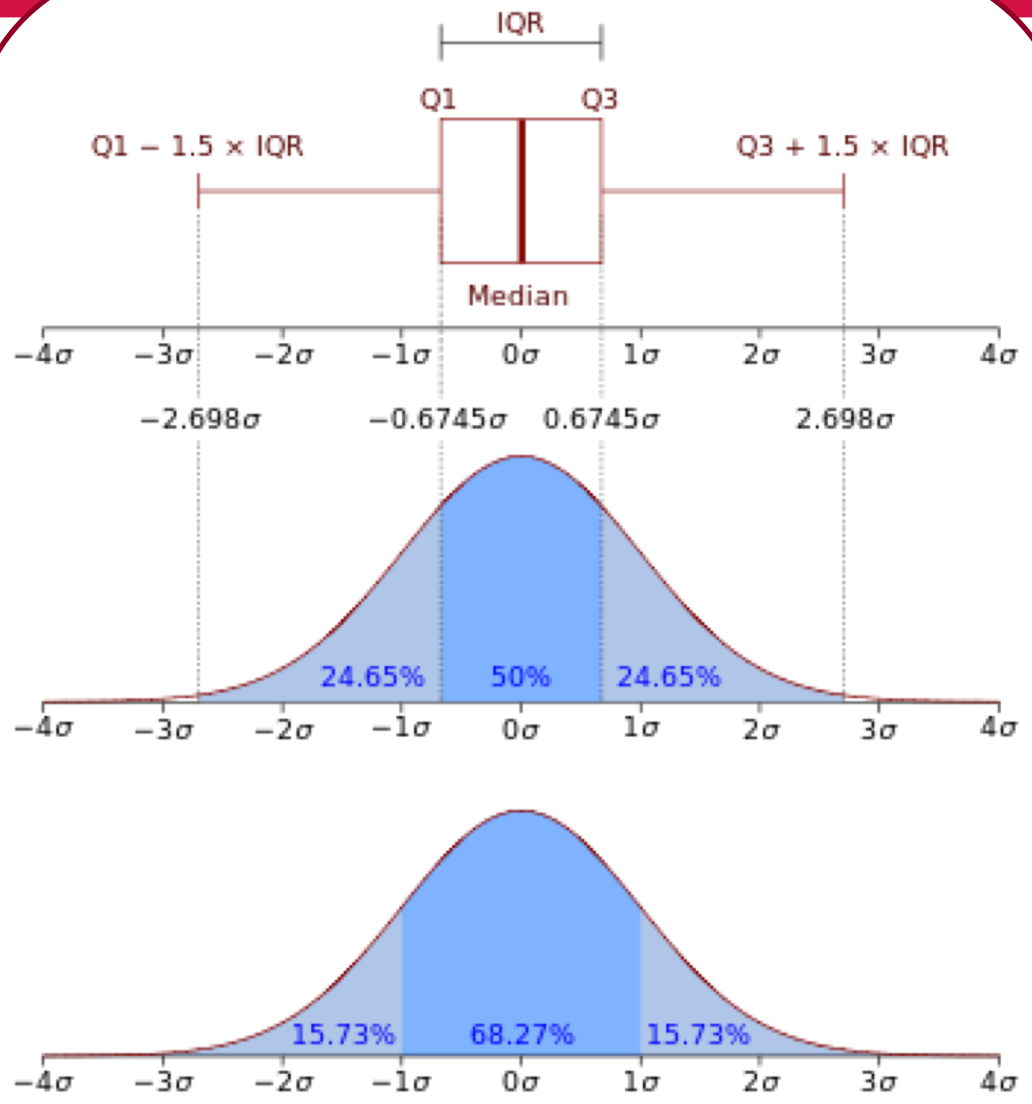
- A random variable is a function that associates a unique numerical value with every outcome of an experiment. The value of the random variable will vary from trial to trial as the experiment is repeated (e.g. randomized quicksort)
- **Examples:**
  - A coin is tossed ten times. The random variable X is the number of tails that are noted. X can only take the values 0, 1, 2, … 10
  - A light bulb is burned until it burns out. The random variable Y is its lifetime in hours. Y can take any positive real value

Probability distribution of a Discrete Random Variables

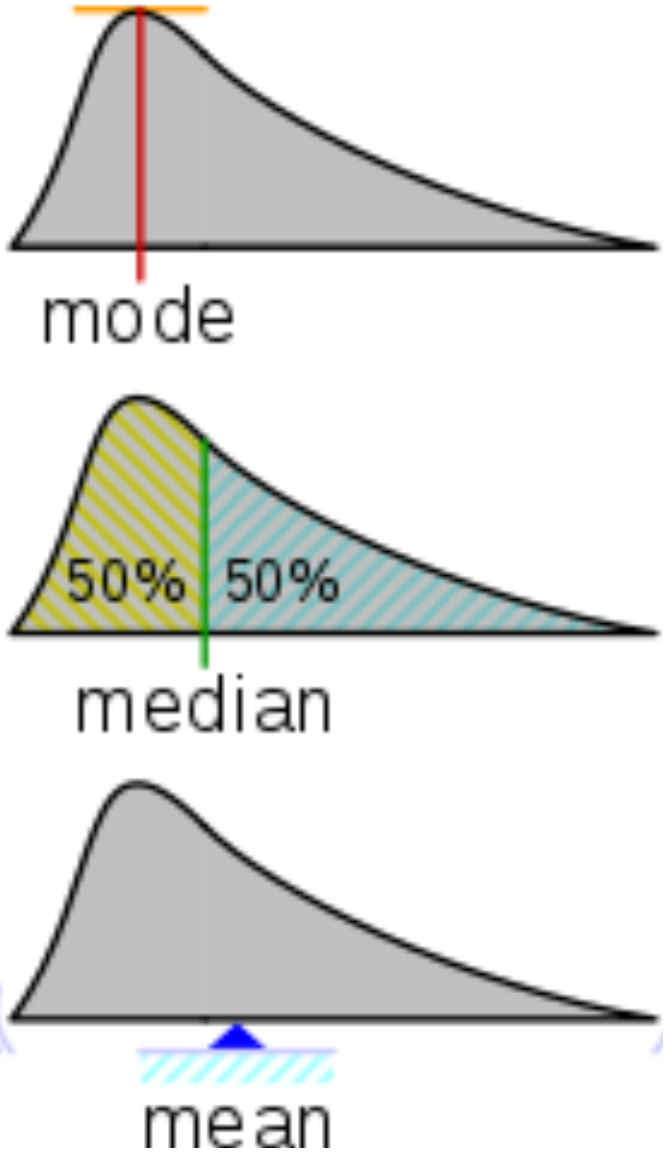
- The probability distribution of a discrete random variable is a list of probabilities associated with each of its possible values
- More formally, let X be a discrete random variable taking on distinct values x_1, x_2, x_n.
- Then the probability density function (PDF) of the random variable X, denoted by p(x) or f(x) as:

$$f(x_i) = \begin{cases} P(X = x_i) & for\ i = 1, 2, \ldots, n, \ldots \\ 0 & for\ x \neq x_i \end{cases}$$

Probability distribution of a Discrete Random Variables



Normal Distribution



Probability Distribution of a Discrete Random Variables

Properties:

1. $f(x_i) \geq 0, \quad for\ all\ i$
2. $\sum_i f(x_i) = 1$

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed

Heads / Tails

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the **CDF**
- Solution S={HHH

Cumulative Distribution Function

Heads

Heads

Heads

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH , **HHT,**

Heads

Heads

Tails

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH , HHT, **HTH**, …

Heads

Tails

Heads

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH, HHT, HTH, THH, HTT, THT, TTH, TTT}
- Let X = number of heads, then x = 0, 1, 2, 3

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH, HHT, HTH, THH, HTT, THT, TTH, TTT}
- Let X = number of heads then x = 0, 1, 2, 3

$$f(0) = P(X = 0) = P\left[\{TTT\}\right] = 1/8$$

1 out of 8

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH, HHT, HTH, THH, HTT, THT, TTH, TTT}
- Let X = number of heads then x = 0, 1, 2, 3

$$f(0) = P(X = 0) = P\left[\{TTT\}\right] = 1/8$$

$$f(1) = P(X = 1) = P\left[\{HTT, THT, TTH\}\right] = 3/8$$

3 out of 8

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH, HHT, HTH, THH, HTT, THT, TTH, TTT}
- Let X = number of heads then x = 0, 1, 2, 3

$$f(0) = P(X = 0) = P\left[\{TTT\}\right] = 1/8$$
$$f(1) = P(X = 1) = P\left[\{HTT, THT, TTH\}\right] = 3/8$$
$$f(2) = P(X = 2) = P\left[\{HHT, HTH, THH\}\right] = 3/8$$

3 out of 8

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH, HHT, HTH, THH, HTT, THT, TTH, TTT}
- Let X = number of heads then x = 0, 1, 2, 3

$$f(0) = P(X = 0) = P\left[\{TTT\}\right] = 1/8$$
$$f(1) = P(X = 1) = P\left[\{HTT, THT, TTH\}\right] = 3/8$$
$$f(2) = P(X = 2) = P\left[\{HHT, HTH, THH\}\right] = 3/8$$
$$f(3) = P(X = 3) = P\left[\{HHH\}\right] = 1/8$$

1 out of 8

Distribution Function - Example

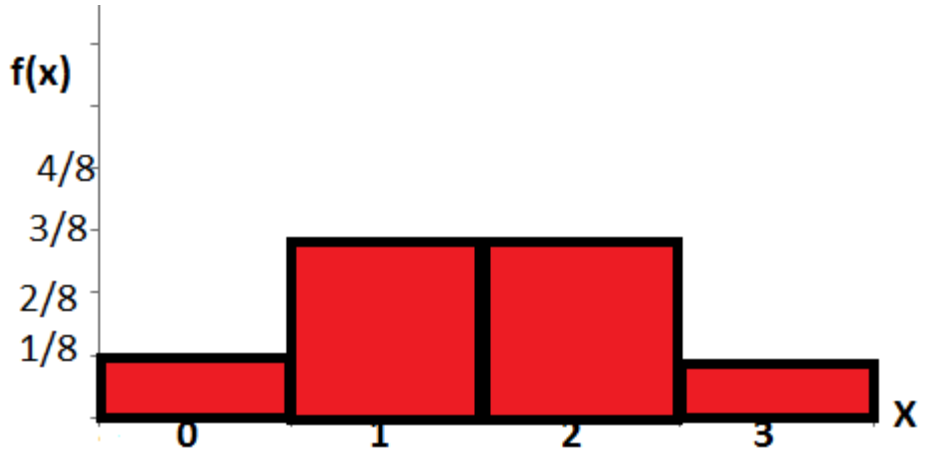
- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH, HHT, HTH, THH, HTT, THT, TTH, TTT}
- Let X = number of heads then x = 0, 1, 2, 3

$$f(0) = P(X = 0) = P\left[\{TTT\}\right] = 1/8$$
$$f(1) = P(X = 1) = P\left[\{HTT, THT, TTH\}\right] = 3/8$$
$$f(2) = P(X = 2) = P\left[\{HHT, HTH, THH\}\right] = 3/8$$
$$f(3) = P(X = 3) = P\left[\{HHH\}\right] = 1/8$$

No of heads (x_i)	Probability $P(x_i)$ or $f(x_i)$
0	1/8
1	3/8
2	3/8
3	1/8
Total	**1**

Distribution Function - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution S={HHH, HHT, HTH, THH, HTT, THT, TTH, TTT}
- Let X = number of heads then x = 0, 1, 2, 3



No of heads (xi)	Probability P(xi) or f(xi)
0	1/8
1	3/8
2	3/8
3	1/8
Total	**1**

Random Variable - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution:
  - The CDF of X is:

Probability of At most 0 Hs

(x_i)	$f(x_i)$	Distribution Function $F(x_i)=P(X<=x_i)$
0	1/8	1/8: P(X<=0)=1/8

Random Variable - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution:
  - The CDF of X is:

Probability of At most 1 H

(x_i)	$f(x_i)$	Distribution Function $F(x_i)=P(X<=x_i)$
0	1/8	1/8: P(X<=0)=1/8
1	3/8	4/8: P(X<=1)=P(X=0)+P(X=1)=1/8+3/8=4/8

Random Variable - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution:
  - The CDF of X is:

Probability of At most 2 Hs

(x_i)	$f(x_i)$	Distribution Function $F(x_i)=P(X<=x_i)$
0	1/8	1/8: P(X<=0)=1/8
1	3/8	4/8: P(X<=1)=P(X=0)+P(X=1)=1/8+3/8=4/8
2	3/8	7/8: P(X<=2)=P(X=0)+P(X=1) +P(X=2)=1/8+3/8+3/8=7/8

Random Variable - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution:
  - The CDF of X is:

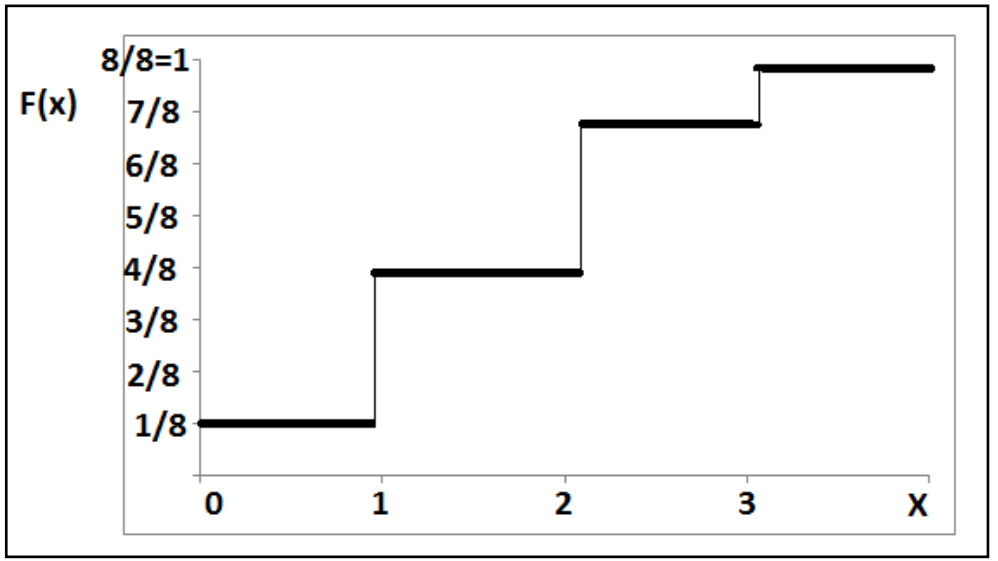
Probability of At most 3 Hs

(x_i)	$f(x_i)$	Distribution Function $F(x_i)=P(X<=x_i)$
0	1/8	1/8: P(X<=0)=1/8
1	3/8	4/8: P(X<=1)=P(X=0)+P(X=1)=1/8+3/8=4/8
2	3/8	7/8: P(X<=2)=P(X=0)+P(X=1) +P(X=2)=1/8+3/8+3/8=7/8
3	1/8	8/8: P(X<=3)=P(X=0)+P(X=1)+P(X=2) +P(X=3)=1/8+3/8+3/8+1/8=8/8=1

Random Variable - Example

- Find the probability distribution and distribution function for the number of heads when 3 balanced coins are tossed
- Construct a probability histogram and a graph of the CDF
- Solution:
  - The CDF of X is:

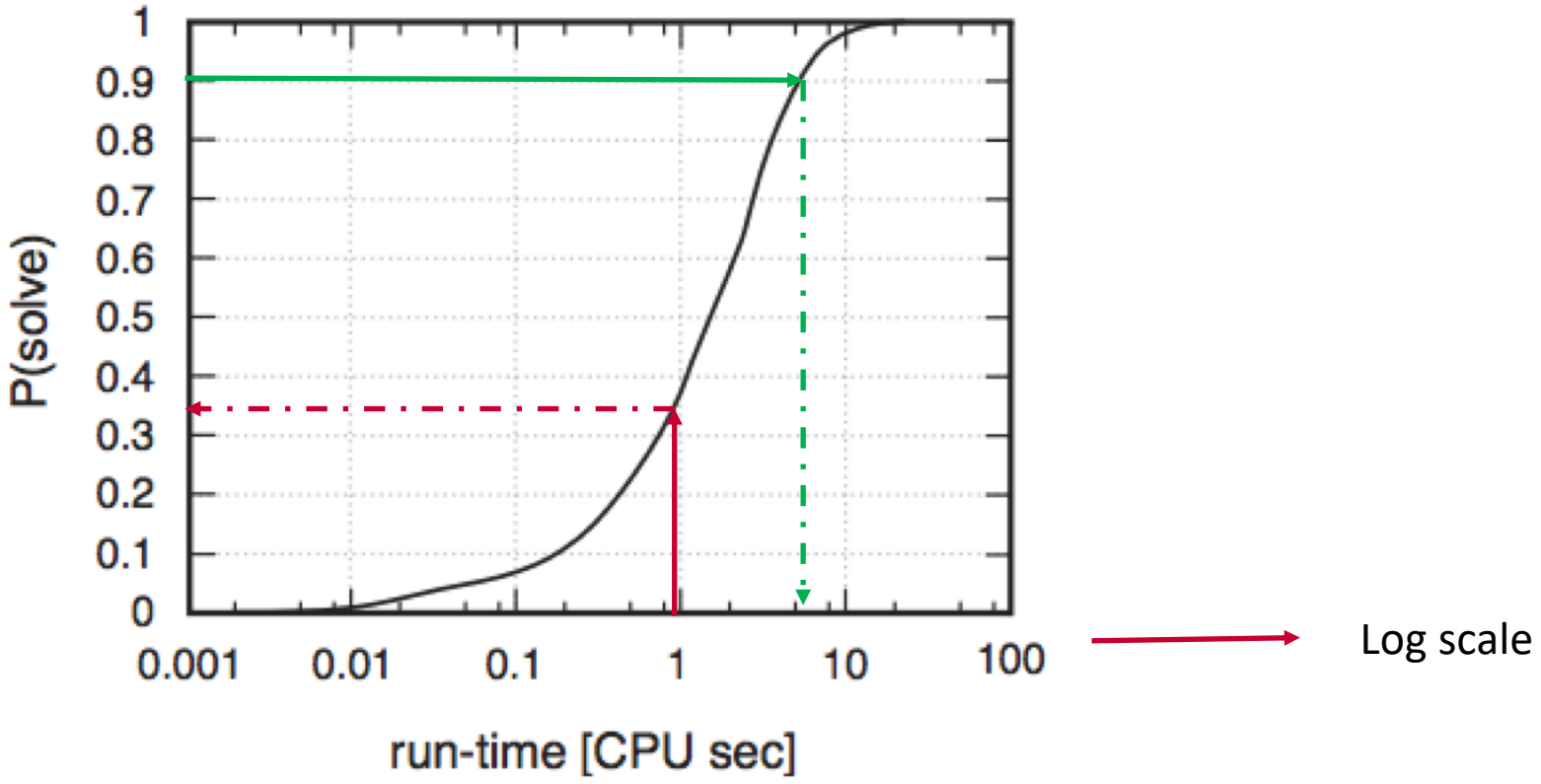
(x_i)	$f(x_i)$	$F(x_i)=P(X<=x_i)$
0	1/8	1/8
1	3/8	4/8
2	3/8	7/8
3	1/8	1



Example of run-time distribution for LS algorithm applied to hard instance of combinatorial decision problem

Runtime for 1000 runs on 100-variable hard 3-SAT instance with WalkSAT!

Random variable:
runtime to solve
a given instance



Run-time Distribution

Given a LS algorithm A for a decision problem $\prod$:

- The success probability $P_s(RT_{A,\pi} \leq t)$ is the probability that A finds a solution for a soluble instance $\boldsymbol{\pi} \in \prod$ in time $\leq t$.
- The run-time distribution (*RTD*) of A on $\boldsymbol{\pi}$ is the probability of the random variable $RT_{A,\boldsymbol{\pi}}$
- The run-time distribution function *rtd*: $\mathbb{R}^+ \rightarrow [0, 1]$, defined as
  $rtd(t) = P_s(RT_{A,\pi} \leq t)$, completely characterizes the RTD of A on $\boldsymbol{\pi}$

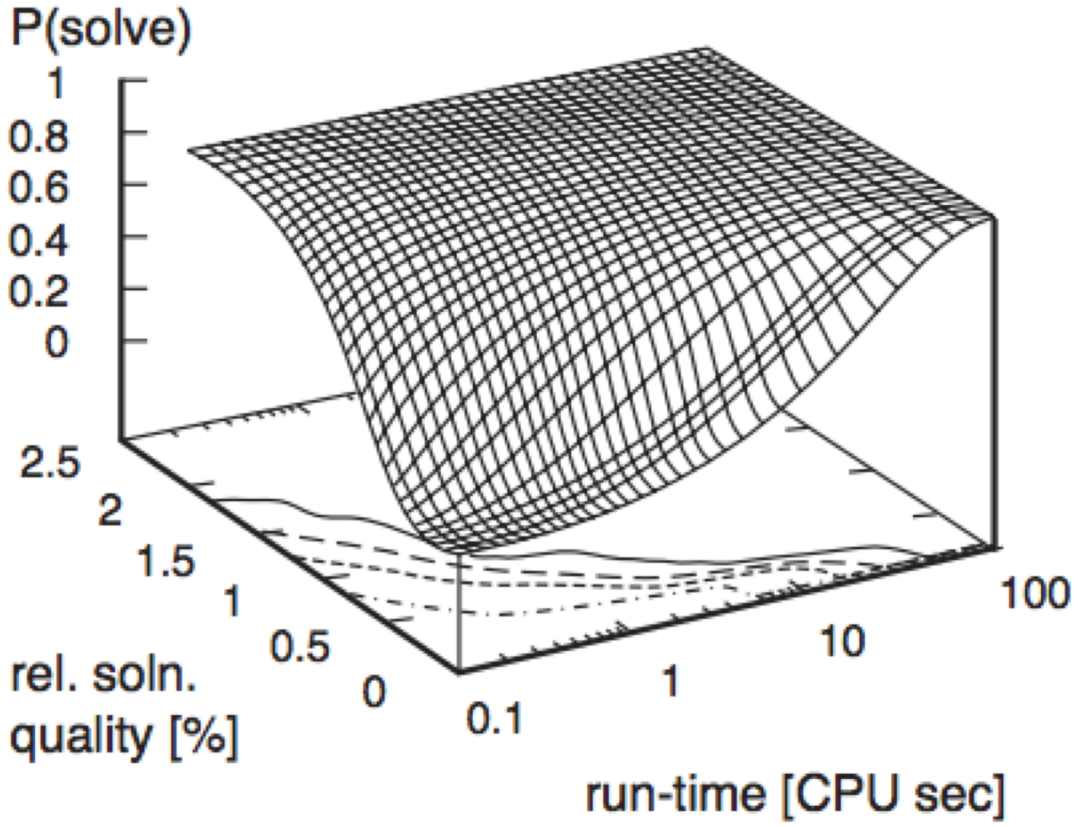
Run-time Distribution

Given a LS algorithm A' for an optimization problem $\prod$':

- The success probability $P_s(RT_{A',\pi'} \leq t, SQ_{A',\pi'} \leq q)$ is the probability that A' finds a solution for a soluble instance $\boldsymbol{\pi}' \in \prod'$ of quality $\leq q$ in time $\leq t$
- The run-time distribution (RTD) of A' on $\boldsymbol{\pi}'$ is the probability of the bivariate random variable $(RT_{A',\pi'}, SQ_{A',\pi'})$
- The run-time distribution function $rtd : \mathbb{R}^+ \times \mathbb{R}^+ \rightarrow [0, 1]$, defined as $rtd(t,q)=P_s(RT_{A',\pi'} \leq t, SQ_{A',\pi'} \leq q)$ completely characterizes the RTD of A' on $\boldsymbol{\pi}'$

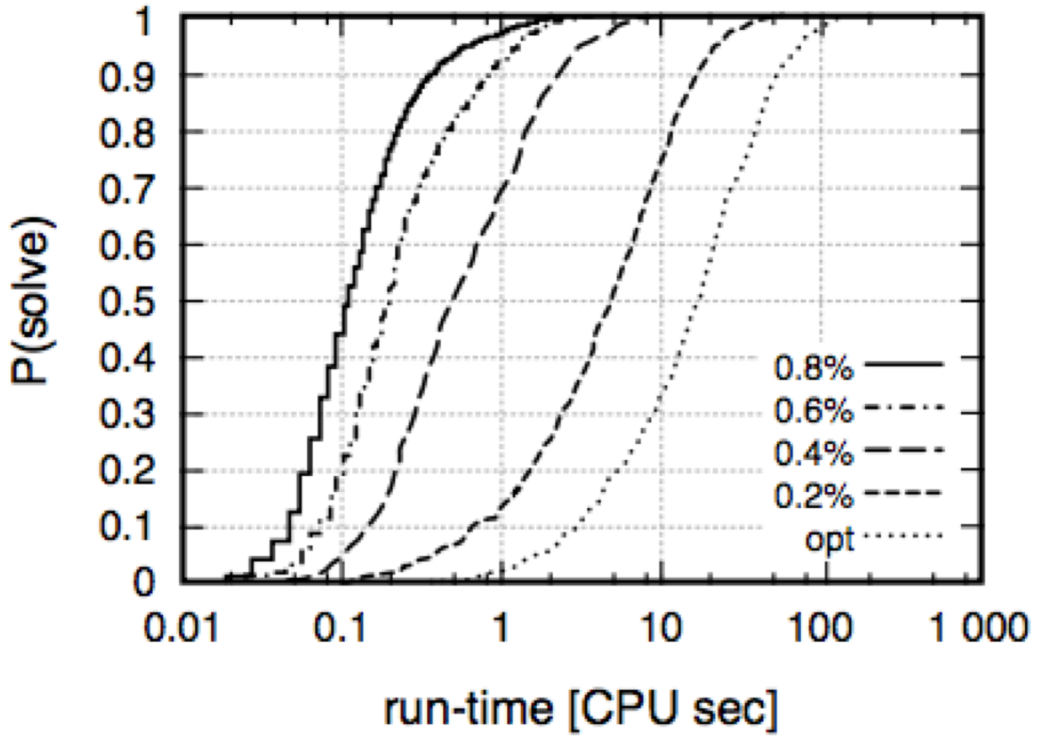
Example of run-time distribution for LS algorithm applied to a hard instance of a combinatorial optimization problem:

Runtime for 1000 runs of an ILS approach on a TSP instance

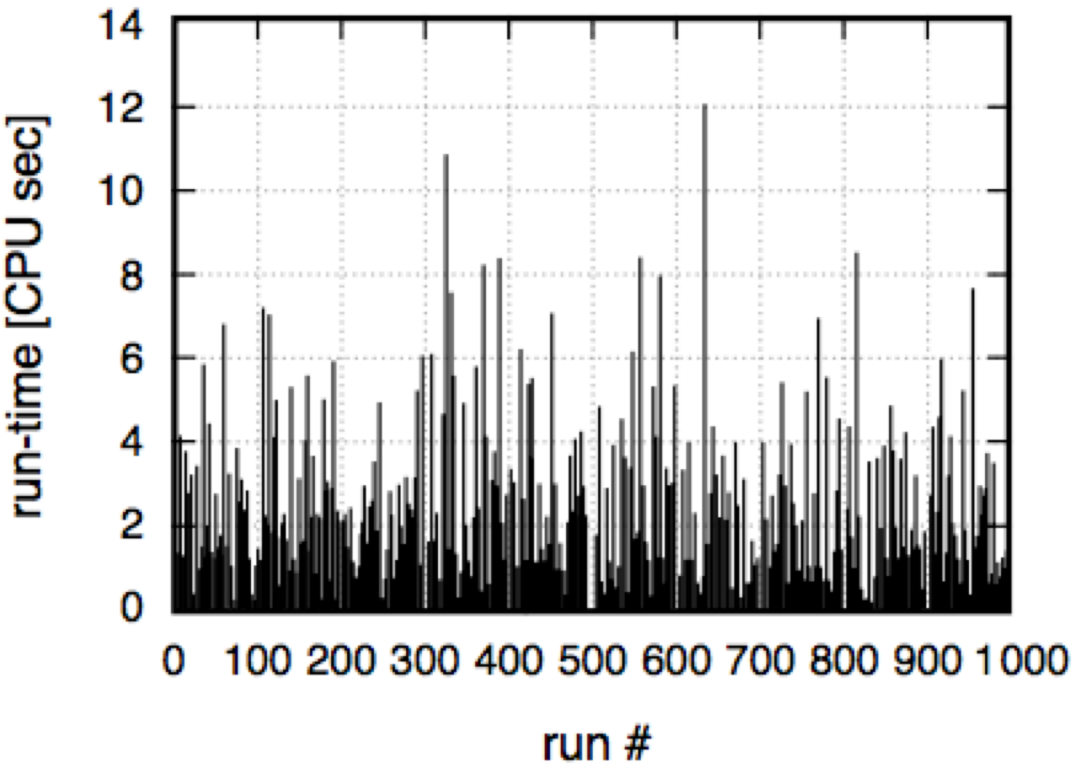


Qualified RTDs for various solution qualities

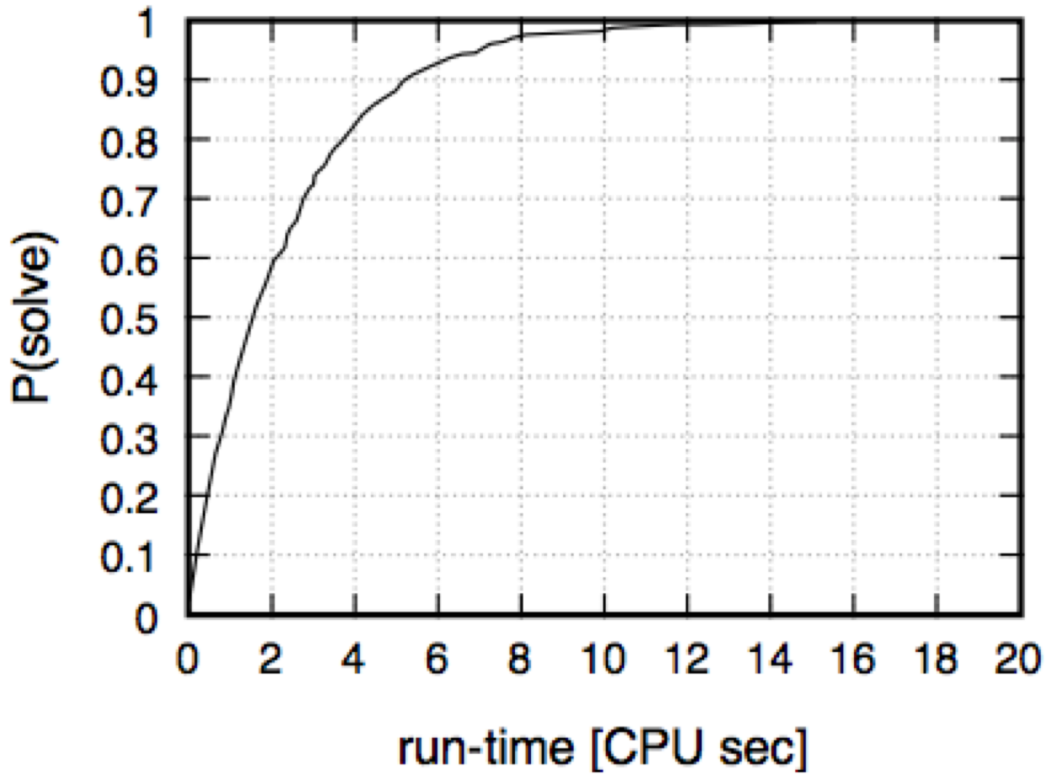
Qualified RTDs: Univariate RTDs for **given values of *q***



Typical sample of run-times for a LS algorithm applied to an instance of a hard decision problem:



Corresponding empirical RTD



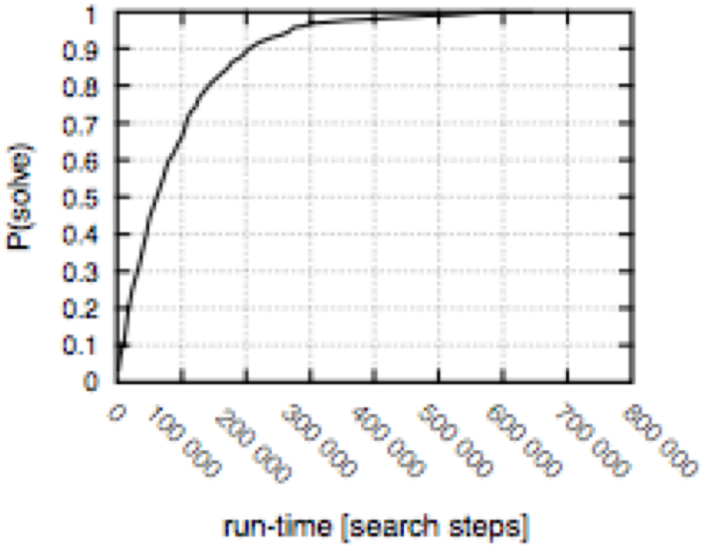
Protocol for obtaining the empirical RTD for a LS algorithm A applied to a given instance π of a decision problem

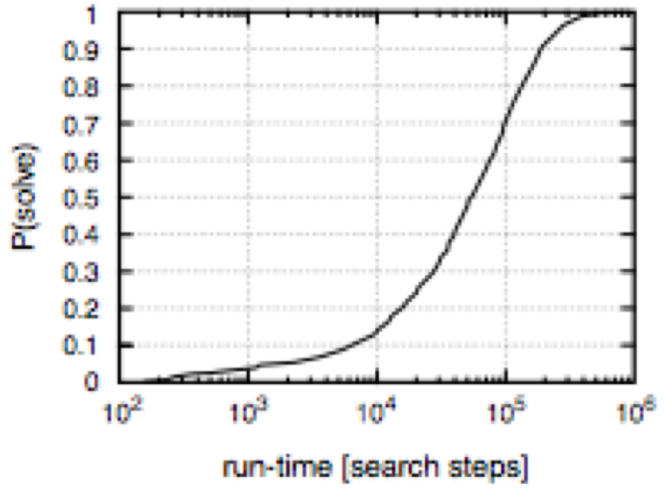
- Perform *k* independent runs of A on π with cutoff time *t'*.
- Record the *k'* successful runs, and for each run, record its run-time in a list *L*
- Sort *L* according to increasing run-time; let *rt(j)* denote the runtime from entry *j* of the sorted list (*j*=1, …, *k'*)
- Plot the graph (*rt(j)*, *j/k*), i.e., the cumulative empirical RTD of A on π

Different views of RTD plots are useful for qualitative analysis of LS behavior

- **Semi-log** plots give a better view of the distribution over its entire range
- Uniform performance differences characterized by a constant factor corresponds to shifts along horizontal axis
- **Log-log plots** of RTD or its associated **failure rate decay function**, *1- rtd(t)*, are often useful for examining behavior for very short or very long runs.

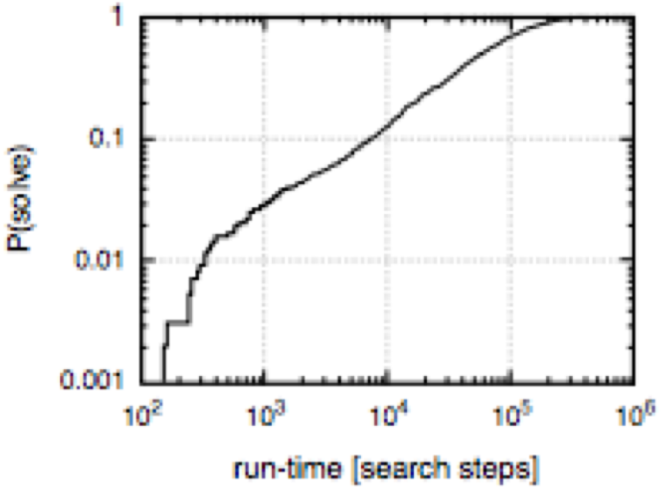
Various graphical representations of a typical RTD

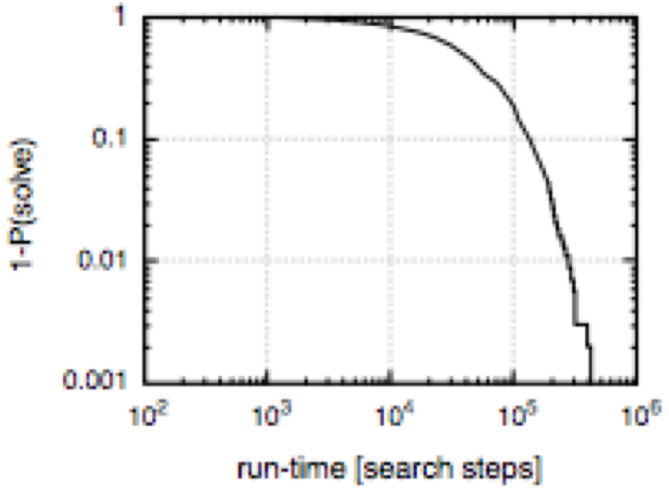




Semi-log

Log-log





Failure rate log-log

Quantitative RTD Analysis

Include basic descriptive statistics, such as:

- Mean
- Median ($q_{0.5}$) and other quantiles (e.g., $q_{0.25}$, $q_{0.75}$, $q_{0.9}$)
- Standard deviation, or (better) the *coefficient of variation (cv)* = stddev/mean

Note: SLS algorithms typically show very high variability in run-time; therefore, reporting a measure of variability along with the mean or median run-time is important

Example

The empirical RTD previously plotted could be characterized by the following basic descriptive statistics

mean	57,606.23	median	38,911
min	107	$q_{0.25}$; $q_{0.1}$	16,762; 5,332
max	443,493	$q_{0.75}$; $q_{0.9}$	80,709; 137,863
stddev	58,953.60	$q_{0.75}/q_{0.25}$	4.81
cv	1.02	$q_{0.9}/q_{0.1}$	25.86

Important

Quantiles (such as the median) are more stable w.r.t. extreme values than the mean but also worth noting how extreme the values can get.

Quite different to have distributions

999,…., 999, 1000, 1001, …., 1001

1,…., 1, 1000, 1000000000, …., 1000000000

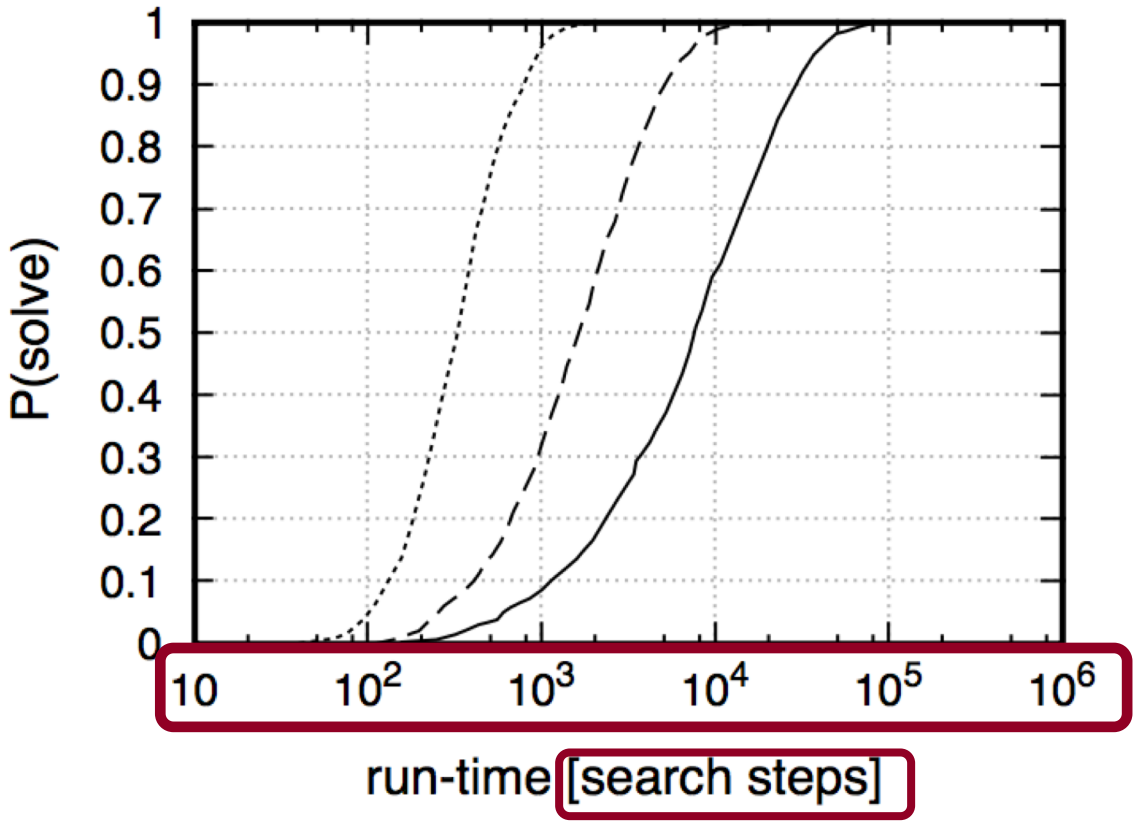
Seconds

Minutes

Hours!!

RTD for WalkSAT on three 3-SAT instances

Instances generated by **same generator** with **same parameters**



Basic quantitative analysis for ensembles of instances

- For bigger sets of instances (e.g., samples from random instance distributions), it is important to characterize the performance of the given algorithm on **individual instances** as well as the **entire ensemble**

- Report and analyse run-time distributions on representative instance(s) as well as **search cost distribution (SCD)**, i.e., distribution of RTD statistics (e.g., mean or median) across a given instance ensemble

- For sets of instances that have been generated by systematically varying a parameter (e.g., problem size), **study RTD characteristics in dependence of the parameter value**

Measuring run-times

- CPU time measurements are based on a specific implementation and run-time environment (machine, operating system) of the given algorithm

- To ensure reproducibility and comparability of empirical results, CPU times should be measured in a way that is as independent as possible from machine load.

- When reporting CPU times, the run-time environment should be specified (at least CPU type, model, speed, cache size; amount of RAM; OS type and version); ideally the implementation of the algorithm should be made available.
  - E.g For my thesis I included "*All experiments, for which runtime results are given, were run on an Intel Xeon 2.66GHz machine with 12GB of RAM on Fedora 9*"

Measuring run-times

- Can use the *time* command in Linux systems

  time python Grimes_1234_WalkSAT.py uf20-01.cnf 100 1000 10 0.4 5

- Outputs *user*, *real* and *sys* time
- **Real** is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (e.g. if it is waiting for I/O to complete).
- **User** is amount of CPU time spent in user-mode code (outside the kernel) ***within* the process**. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel **within the process**. This means executing CPU time spent in system calls *within the kernel,* as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process.
- *User+Sys* will tell you how much actual CPU time your process used. Note that this is across all CPUs, so if the process has multiple threads (and this process is running on a computer with more than one processor) it could potentially exceed the wall clock time reported by Real (which usually occurs).

Measuring run-times

To achieve better abstraction from the implementation and run-time environment, it is often preferable to measure run-time using:

- **Operation counts** that reflect the number of operations that contribute significantly towards an algorithm performance
- **Cost models** that specify the CPU time for each such operation for a given implementation and run-time environment

Example

For a given LS algorithm for SAT applied to a specific SAT instance we observe

- A median run-time of 38911 search steps (operation count);
- The CPU time required for each search step is 0.027 ms, while the initialization takes 0.8ms (cost model) -- when running the algorithm on an Intel Xeon 2.4 GHz CPU with 515KB of cache and 1GB RAM running Red Hat Linux, Version 2.4spm (run-time environment)

Run-length distributions:

- RTDs based on run-times measured in terms of elementary operations of the given algorithm are also called **run-length distributions (RLDs)**
- **Caution:** RLDs should be based on elementary operations that either require constant CPU time (for the given problem instance), or on aggregate counts in which operations that require different amounts of CPU time (e.g., two types of search steps) are weighted appropriately
- Elementary operations commonly used as the basis for RLD and other run-time measurements of SLS algorithms include search steps, objective function evaluations and updates of data structures used for implementing the step function

Satisfiability Problems

Material based on KNOWLEDGE REPRESENTATION & REASONING - **SAT**
www.icsd.aegean.gr › lecturers › konsterg › teaching › SATLS
www.icsd.aegean.gr › lecturers › konsterg › teaching › SATDPLL

The Boolean Satisfiability Problem (SAT)

- **Tractable subclasses**
  - Horn-SAT
  - 2-SAT
- **CNF**

Systematic search
Not in this module

- **Algorithms for SAT**
  - DPLL-based
    - Basic chronological backtracking algorithm
    - Branching heuristics
    - Look-ahead (propagation)
    - Backjumping and learning
    - SOA: **Conflict-Driven Clause Learning** solvers

    (solving instances with several million variables and clauses!)
  - Local Search
    - GSAT
    - WalkSAT
    - Other enhancements

Aside: Nice paper on CDCL approach

https://www.princeton.edu/~chaff/publication/DAC2001v56.pdf

Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz
Department of EECS
UC Berkeley
moskewcz@alumni.princeton.edu

Conor F. Madigan
Department of EECS
MIT
cmadigan@mit.edu

Ying Zhao, Lintao Zhang, Sharad Malik
Department of Electrical Engineering
Princeton University
{yingzhao, lintaoz, sharad}@ee.princeton.edu

ABSTRACT

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in Artificial Intelligence (AI). This study has culminated in the development of several SAT packages, both proprietary and in the public domain (e.g. GRASP, SATO) which find significant use in both research and industry. Most existing complete solvers are variants of the Davis-Putnam (DP) search algorithm. In this paper we describe the development of a new complete solver, Chaff, which achieves significant performance gains through careful engineering of all aspects of the search – especially a particularly efficient implementation of Boolean constraint propagation (BCP) and a novel low overhead decision strategy. Chaff has been able to obtain one to two orders of magnitude performance improvement on difficult SAT benchmarks in comparison with other solvers (DP or otherwise), including GRASP and SATO.

Many publicly available SAT solvers (e.g. GRASP [8], POSIT [5], SATO [13], rel_sat [2], WalkSAT [9]) have been developed, most employing some combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. Heuristic local search techniques are not guaranteed to be complete (i.e. they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability); as a result, complete SAT solvers (including ours) are based almost exclusively on the DP search algorithm.

1.1 Problem Specification

Most solvers operate on problems for which f is specified in conjunctive normal form (*CNF*). This form consists of the logical AND of one or more *clauses*, which consist of the logical OR of one or more *literals*. The *literal* comprises the fundamental logical unit in the problem, being merely an instance of a variable or its complement. (In this paper, complement is represented by $\neg$.) All Boolean functions can be described in the *CNF* format. The advantage of *CNF* is that in this form, for f to be satisfied (*sat*), each individual *clause* must be *sat*.

Conjunctive Normal Form (CNF)

- Variables vs literals

- A formula A is in conjunctive normal form, or simply CNF, if it is
  - either *T*, or *F*, or a conjunction of disjunctions of **literals**:

$$\bigwedge_i \bigvee_j (x_{i,j})$$

i.e. a conjunction of clauses.

Conjunctive Normal Form

- Every sentence in propositional logic can be transformed into *conjunctive normal form*
  - i.e. a conjunction of disjunctions

Conversion steps

1. Eliminate $\Rightarrow$ using the rule that $(p \Rightarrow q)$ is equivalent to $(\neg p \vee q)$

2. Use de Morgan's laws so that negation applies to literals only

3. Distribute $\vee$ and $\wedge$ to write the result as a conjunction of disjunctions

Conjunctive Normal Form - Example

Convert to CNF $\neg(p \Rightarrow q) \vee (r \Rightarrow p)$

1. Eliminate implication signs
   - $\neg(\neg p \vee q) \vee (\neg r \vee p)$
2. Apply de Morgan's laws
   - $(p \wedge \neg q) \vee (\neg r \vee p)$
3. Apply associative and distributive laws
   - $(p \vee \neg r \vee p) \wedge (\neg q \vee \neg r \vee p)$
   - $(p \vee \neg r) \wedge (\neg q \vee \neg r \vee p)$

Uninformed random walk for SAT

```
procedure URW-for-SAT(F, maxSteps)
    input: propositional formula F, integer maxSteps
    output: model of F or ∅
    choose assignment a of truth values to all variables in F
        uniformly at random;
    steps := 0;
    while not((a satisfies F) and (steps < maxSteps)) do
        randomly select variable x in F;
        change value of x in a;
        steps := steps+1;
    end
    if a satisfies F then
        return a
    else
        return ∅
    end
end URW-for-SAT
```

Search space S: set of all truth assignment to variables in a given formula F

Solution set S': set of all models of F

Neighborhood relation N: 1-flip neighborhood, i.e.,

assignments are neighbors in N iff they differ in the truth

value of exactly one variable

Uninformed random walk for SAT

```
procedure URW-for-SAT(F, maxSteps)
    input: propositional formula F, integer maxSteps
    output: model of F or ∅
    choose assignment a of truth values to all variables in F
        uniformly at random;
    steps := 0;
    while not((a satisfies F) and (steps < maxSteps)) do
        randomly select variable x in F;
        change value of x in a;
        steps := steps+1;
    end
    if a satisfies F then
        return a
    else
        return ∅
    end
end URW-for-SAT
```

Uniform random choice from S

When a "*model*" or solution is found

random choice from S from current neighborhood

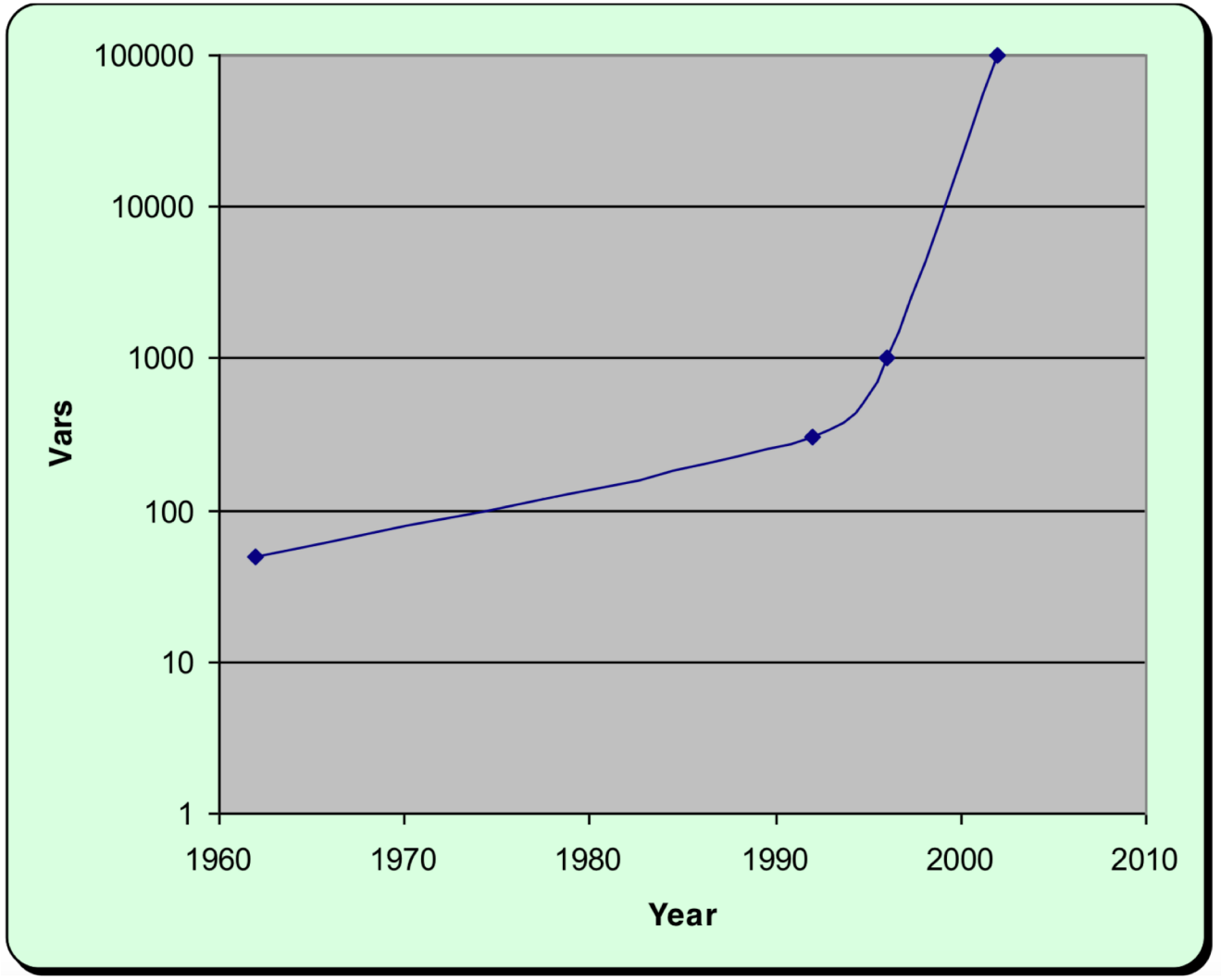
These uninformed SLS strategies are quite ineffective, but play a role in combination with more directed search strategies

Local Search for SAT

- Despite the success of modern DPLL-based solvers on real problems, there are still very large (and hard) instances that are out of their reach
  - Some random SAT problems really are hard!

- Local search methods are very successful on a number of such instances
  - they can quickly find models, if models exist
  - …but cannot prove insolubility if no models exists

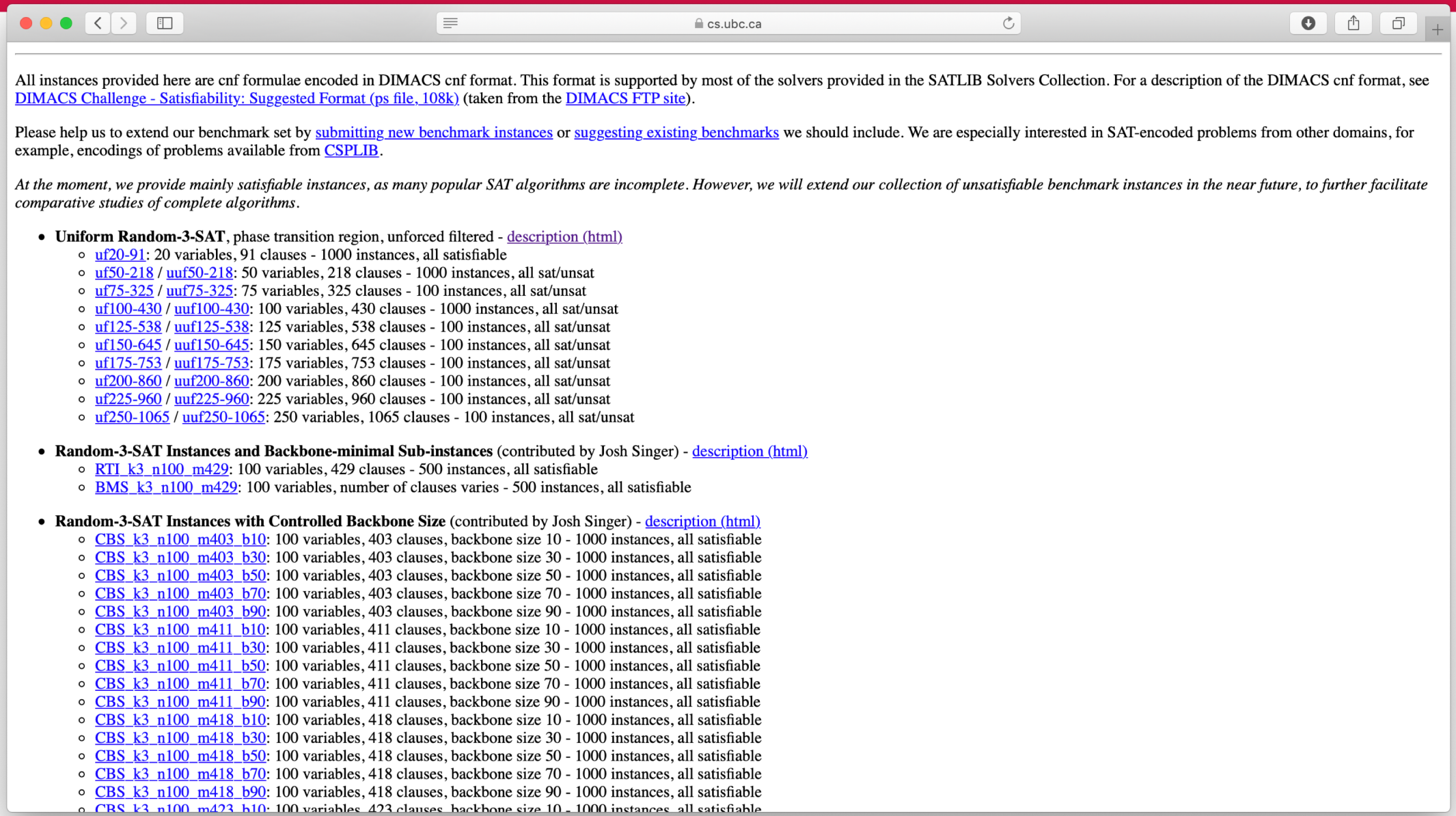
SAT Increase in instance sizes solvable



k-SAT

- Subclass of the SAT problem
  - Exactly k variables in each clause
  - $(P \vee \neg Q) \wedge (Q \vee R) \wedge (\neg R \vee \neg P)$ is in 2-SAT
- 3-SAT is NP-complete
  - SAT can be reduced to 3-SAT
- 2-SAT is in P
  - Exists linear time algorithm

SATLib



All instances provided here are cnf formulae encoded in DIMACS cnf format. This format is supported by most of the solvers provided in the SATLIB Solvers Collection. For a description of the DIMACS cnf format, see DIMACS Challenge - Satisfiability: Suggested Format (ps file, 108k) (taken from the DIMACS FTP site).

Please help us to extend our benchmark set by submitting new benchmark instances or suggesting existing benchmarks we should include. We are especially interested in SAT-encoded problems from other domains, for example, encodings of problems available from CSPLIB.

At the moment, we provide mainly satisfiable instances, as many popular SAT algorithms are incomplete. However, we will extend our collection of unsatisfiable benchmark instances in the near future, to further facilitate comparative studies of complete algorithms.

- **Uniform Random-3-SAT**, phase transition region, unforced filtered - description (html)
  - uf20-91: 20 variables, 91 clauses - 1000 instances, all satisfiable
  - uf50-218 / uuf50-218: 50 variables, 218 clauses - 1000 instances, all sat/unsat
  - uf75-325 / uuf75-325: 75 variables, 325 clauses - 100 instances, all sat/unsat
  - uf100-430 / uuf100-430: 100 variables, 430 clauses - 1000 instances, all sat/unsat
  - uf125-538 / uuf125-538: 125 variables, 538 clauses - 100 instances, all sat/unsat
  - uf150-645 / uuf150-645: 150 variables, 645 clauses - 100 instances, all sat/unsat
  - uf175-753 / uuf175-753: 175 variables, 753 clauses - 100 instances, all sat/unsat
  - uf200-860 / uuf200-860: 200 variables, 860 clauses - 100 instances, all sat/unsat
  - uf225-960 / uuf225-960: 225 variables, 960 clauses - 100 instances, all sat/unsat
  - uf250-1065 / uuf250-1065: 250 variables, 1065 clauses - 100 instances, all sat/unsat

- **Random-3-SAT Instances and Backbone-minimal Sub-instances** (contributed by Josh Singer) - description (html)
  - RTI_k3_n100_m429: 100 variables, 429 clauses - 500 instances, all satisfiable
  - BMS_k3_n100_m429: 100 variables, number of clauses varies - 500 instances, all satisfiable

- **Random-3-SAT Instances with Controlled Backbone Size** (contributed by Josh Singer) - description (html)
  - CBS_k3_n100_m403_b10: 100 variables, 403 clauses, backbone size 10 - 1000 instances, all satisfiable
  - CBS_k3_n100_m403_b30: 100 variables, 403 clauses, backbone size 30 - 1000 instances, all satisfiable
  - CBS_k3_n100_m403_b50: 100 variables, 403 clauses, backbone size 50 - 1000 instances, all satisfiable
  - CBS_k3_n100_m403_b70: 100 variables, 403 clauses, backbone size 70 - 1000 instances, all satisfiable
  - CBS_k3_n100_m403_b90: 100 variables, 403 clauses, backbone size 90 - 1000 instances, all satisfiable
  - CBS_k3_n100_m411_b10: 100 variables, 411 clauses, backbone size 10 - 1000 instances, all satisfiable
  - CBS_k3_n100_m411_b30: 100 variables, 411 clauses, backbone size 30 - 1000 instances, all satisfiable
  - CBS_k3_n100_m411_b50: 100 variables, 411 clauses, backbone size 50 - 1000 instances, all satisfiable
  - CBS_k3_n100_m411_b70: 100 variables, 411 clauses, backbone size 70 - 1000 instances, all satisfiable
  - CBS_k3_n100_m411_b90: 100 variables, 411 clauses, backbone size 90 - 1000 instances, all satisfiable
  - CBS_k3_n100_m418_b10: 100 variables, 418 clauses, backbone size 10 - 1000 instances, all satisfiable
  - CBS_k3_n100_m418_b30: 100 variables, 418 clauses, backbone size 30 - 1000 instances, all satisfiable
  - CBS_k3_n100_m418_b50: 100 variables, 418 clauses, backbone size 50 - 1000 instances, all satisfiable
  - CBS_k3_n100_m418_b70: 100 variables, 418 clauses, backbone size 70 - 1000 instances, all satisfiable
  - CBS_k3_n100_m418_b90: 100 variables, 418 clauses, backbone size 90 - 1000 instances, all satisfiable
  - CBS_k3_n100_m423_b10: 100 variables, 423 clauses, backbone size 10 - 1000 instances, all satisfiable