Runtime Complexity of Golomb Ruler as a Puzzle: Self-Avoiding Walks and Solver Instrumentation via xBed

Franc Brglez	Janez Brest	Borko Bošković
Computer Science	Elec. Eng. & Comp. Science	Elec. Eng. & Comp. Science
NC State University	University of Maribor	University of Maribor
Raleigh, NC 27695, USA	SI-2000 Maribor, Slovenia	SI-2000 Maribor, Slovenia
brglez@ncsu.edu	janez.brest@um.si	borko.boskovic@um.si

Summary. Solving combinatorial problems as puzzles has been illustrated with a version of lights-out puzzle in the biographical movie Beautiful Mind. Such problems are also known as *NP optimization problems (NPOPs)*; their formal description corresponds to a 4-tuple *(I, sol, m, goal)*[1]. We argue that the formal description of a puzzle is the other side of the same coin: I is the set of puzzle instances, $sol(x) = y$ is the set of feasible solutions for instance $x \in I$, $m(x, y)$ is called the *objective function*, and the *goal* is finding the minimum or the maximum of the objective function. There are $O(300)$ such problems (or puzzles) categorized to date[2]. The challenge for theoreticians is to improve *approximability* of hard problems; the challenge for experimentalists is to improve, with statistical significance, *the average-case asymptotic performance* of heuristic methods. This article focuses on the latter: we revisit the optimum Golomb ruler problem[3], formulate a number of new stochastic search strategies and report on scalable average-case asymptotic performance experiments that predict average runtime performance as the problem size increases. The Appendix highlights features of the generic testbed environment (xBed) that not only supports instrumentation for performance experiments of combinatorial solvers but also the methodology to make such experiments reproducible.

The number of choices of stochastic search algorithms is overwhelming[4]. In our approach, we map a puzzle onto a data structure where the concept of *a walk*, one-step-at-a-time, is an effective primitive operation to solve the puzzle. The notion of a walk bypasses *the complex parameterization process that is mandatory* under annealing / genetic / evolutionary / memetic / differential / ant / bat / cuckoo / frog / tabu / stochastic search metaphors. Instead, critical decisions are rooted in (1) problem-specific data structures and (2) statistics fundamentals, including the concept of the *self-avoiding walk (SAW)*. The implicit data structure is the *Hasse graph*, an extension of the *Hasse diagram*[5]. The terminology of the Hasse graph defines *coordinate types* (from k-ary to permutation) for any objective function, it categorizes *coordinate ranks* and *coordinate neighborhoods*, and formulates any number of walks, from random walks that typically have a number of cycles, to Hamiltonian walks which are self-avoiding by definition, and most importantly, to well-controlled contiguous and long self-avoiding walks that efficiently and effectively search for the coordinate that satisfies the specified goal, defined by the puzzle's objective function.

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial optimization problems and their approximability properties*. Springer Verlag, 1999

[2] See the links under `http://www.nada.kth.se/~viggo/problemlist/compendium.html`

[3] See `http://en.wikipedia.org/wiki/Golomb_ruler`

[4] Each of the root keywords in this table is extended with '`search`' and '`algorithm`' before quering the Google search engine on August 15, 2015.

root keywords	number-of-hits
annealing	2,430,000
genetic	28,100,000
evolutionary	1,780,000
memetic	262,000
differential	2,890,000
ant	2,100,000
bat	525,000
cuckoo	123,000
frog	477,000
tabu	924,000
etc	82,600,000

[5] `http://en.wikipedia.org/wiki/Hasse_diagram`

1 Introduction

THE EXTENSION OF HASSE DIAGRAM to a vertex-labeled Hasse graph[6] represents a combinatorial puzzle as a data structure where the concept of a walk, one-step-at-a-time, becomes an effective primitive operation to solve the puzzle. Two solver prototypes, each adapting a *self-avoiding walk strategy* (SAW), have shown significant promise: one on the space represented by concatenation of $2^n \times 3^{n-1}$ binary/ternary coordinates to solve a 2D protein folding problem[7], another applied to the notoriously hard low binary autocorrelation sequences (LABS) problem[8], significantly outperforming a state-of-the-art tabu/memetic solver.

Two companion papers on asymptotic performance of combinatorial search under simple self-avoiding walks, under binary and permutation coordinates, provide additional arguments for formulating and solving combinatorial problems as puzzles[9,10].

THIS PAPER is motivated by challenges represented by the on-going effort on the computation of the optimal Golomb ruler pairs (m, n): the last two pairs, (26, 462) and (27, 553), were found in 2009 and 2014, respectively[11]. On the other hand, clever constructions for such rulers were devised much earlier[12], results plotted in Figure 1 demonstrate that high quality rulers can be calculated in few seconds – but without the proof of optimality. So why continue with the stochastic search for optimum Golomb ruler (ogr) solvers[13,14]? The answer may be quoted from a story about a wise blind man who was asked why he wanted to climb a high mountain – his answer was *"because it is there"*.

Our goal is to prototype a stochastic ogr solver whose asymptotic performance reliably dominates other solvers so that we can argue that, with massive paralellism, this solver may accelerate the current efforts that are being posted on the web[11]. As illustrated in the last section, this solver cannot prove that the solution it is the optimum, but by re-solving the instance with different inital seeds many times *and* without getting censored by runtime constraints, we can argue that solution found is an optimum solution *almost surely*.

The paper is organized as follows. Section 2 introduces the Golomb rules as a puzzle with pegs. Section 3 summarizes notation and definitions, concluding with a simple self-avoding walk saw search algorithm. The puzzle metaphor is particularly useful to also illustrate an efficient *tableau* structure that computes function values adjacent to the reference coordinate very efficiently. Section 4 presents results asymptotic performance experiments. Section 5 uses the solver asymptotic runtime prediction model to project computational requirements as the order of the Golomb ruler increases.

[6] Franc Brglez. Of n-dimensional Dice, Combinatorial Optimization, and Reproducible Research: An Introduction. *Eletrotehniški Vestnik 78(4) 2011: pp. 181–192*, http://ev.fe.uni-lj.si/4-2011/Brglez.pdf, 78(4):181–192, 2011

[7] Franc Brglez. Self-Avoiding Walks across n-Dimensional Dice and Combinatorial Optimization: An Introduction. *Informacije MIDEM, 44 (1) (2014), pp. 53-68*, http://www.midem-drustvo.si/journal/, 44(1):53–68, 2014

[8] Borko Bošković, Franc Brglez, and Janez Brest. Low-Autocorrelation Binary Sequences: On Improved Merit Factors and Runtime Predictions to Achieve Them. http://arxiv.org/, *also under journal review*, 2015

[9] Franc Brglez, Johnny Nguyen, and Yang Ho. Lights-out Problem and the Asymptotic Performance of Combinatorial Search under Self-Avoding Walks. http://arxiv.org/, *also under journal review*, 2015

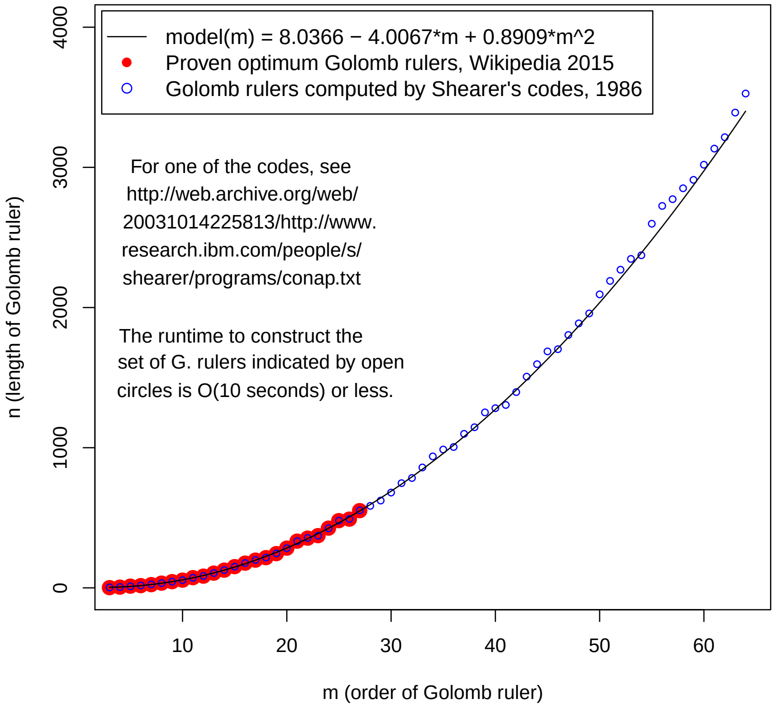
[10] Franc Brglez, Yang Ho, and Johnny Nguyen. Linear-Ordering Problem and the Asymptotic Performance of Combinatorial Search under Self-Avoding Walks. http://arxiv.org/, *also under journal review*, 2015

[11] Golomb Ruler Wikipedia. http://en.wikipedia.org/wiki/Golomb_ruler, March 2015

[12] M. D. Atkinson and Nicola Santoro. Integer Sets with Distinct Sums and Differences and Carrier Frequency Assignments for Nonlinear Repeaters. *IEEE Trans. on Comm.*, 1986

[13] C. Cotta, . Dotu, A. J. Fernandez, and P. Van Hentenryck. A Memetic Approach to Golomb Rulers. In T. P. Runarsson et al, editor, *Parallel Problem Solving from Nature*, LNCS, pages 252–261. Springer, 2007

[14] M.M. Polash, A. Newton, M.A. Hakim, and A. Sattar. Constraint-Based Local Search for Golomb Rulers. In Michel Laurent, editor, *Integration of AI and OR Techniques in Constraint Programming*, LNCS, pages 322–331. Springer, 2015



For one of the codes, see http://web.archive.org/web/20031014225813/http://www.research.ibm.com/people/s/shearer/programs/conap.txt

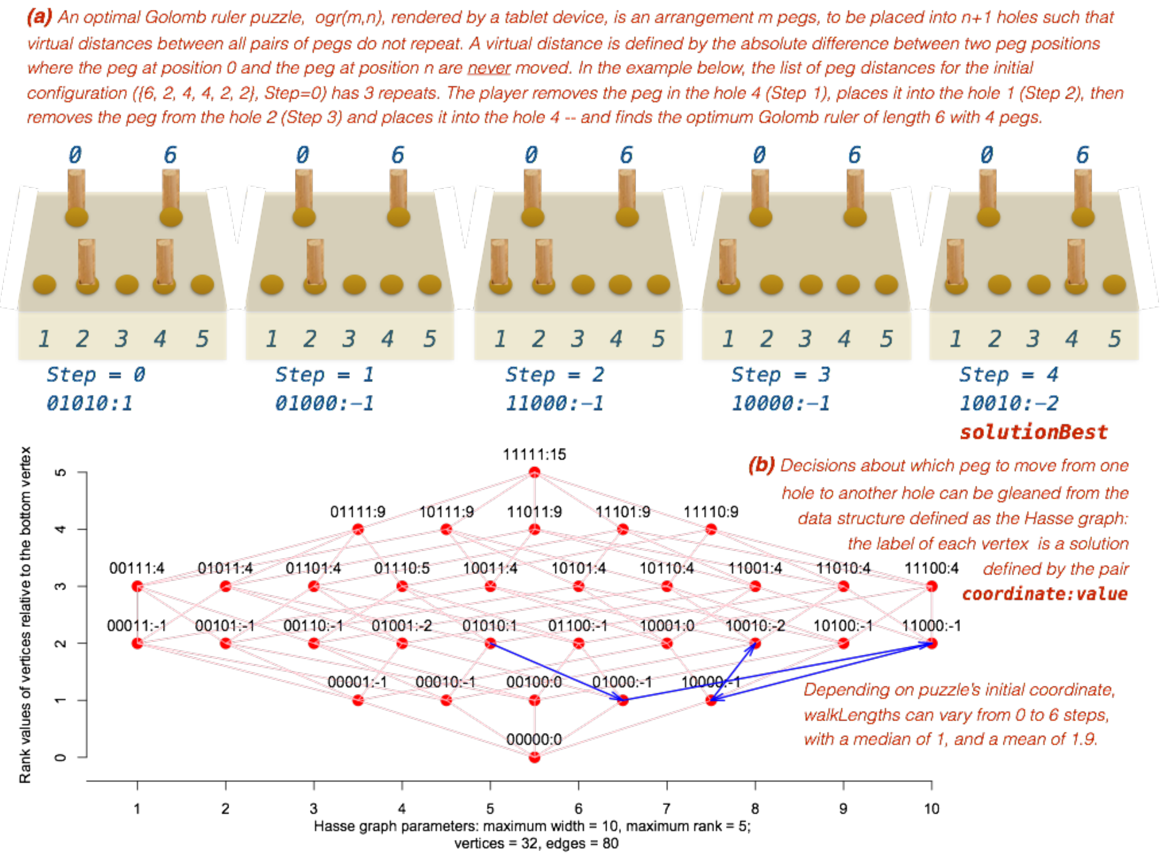
The runtime to construct the set of G. rulers indicated by open circles is O(10 seconds) or less.

Figure 1: A relationship between ruler pairs (m, n): m is the order of the ruler (or the number of marks on the ruler), n is the length of the ruler. This plot shows the shortest known length n of Golomb rulers versus the number of marks m. The solid red circles represent the rulers for which the lengths have been proven as being optimum: the last two pairs, (26, 462) and (27, 553), were found in 2009 and 2014. For the order of $m = 28$, the computational effort to prove that no better ruler exists than the pair $28, 585$, computed by one of Shearer's codes in 1986, is ongoing and is expected to last several years.

2 *Golomb Ruler as a Puzzle*

The Golomb ruler problem (m_{max}, n) can be translated to a puzzle using movable pegs as illustrated for $(6, 4)$ in Figure 2. The quality of arrangement of $(m_{max} - 2)$ pegs in the positions from `(1 2 3 ... n-1)` is measured by computing pairwise distances $|i - j|$ between $m \leq m_{max}$ pegs. If the list of $m(m-1)/2$ distances has 0 repetitions, we say that the ruler with m pegs is feasible for length n. A feasible ruler with $m = m_{max}$ is denoted as the *optimum Golomb ruler*. Figure 2 shows a 4-step solution: the list of all pairwise distances is unique: `(1 2 3 4 5 6)`. We report the solution of the ruler either in terms of peg positions as `(0 1 4 6)` or equivalently, in terms of binary coordinates as `1100101`.

We use a Hasse graph to illustrate the self-avoiding walk solution. Since the peg 0 and the peg 6 are in fixed positions, the binary coordinates in Hasse graph have been trimmed by 1-bit on each side. In the example, the walk starts at the vertex with label `01010:3` and continues under the *meandering walk strategy* (meandering from rank 3 to rank 2 and back) before reaching the solution `10010:0`. However, there are *at least two more self-avoding walk strategies* we could have formulated: (1) *wandering walk* that is less restrictive and is described in Figure 4, or (2), a *reversing walk* that initialize the walk at rank 0 and proceed with $m_{max} - 2$ steps to the coordinate at rank $m_{max} - 2$. If tar-



(a) *An optimal Golomb ruler puzzle, ogr(m,n), rendered by a tablet device, is an arrangement m pegs, to be placed into n+1 holes such that virtual distances between all pairs of pegs do not repeat. A virtual distance is defined by the absolute difference between two peg positions where the peg at position 0 and the peg at position n are never moved. In the example below, the list of peg distances for the initial configuration ({6, 2, 4, 4, 2, 2}, Step=0) has 3 repeats. The player removes the peg in the hole 4 (Step 1), places it into the hole 1 (Step 2), then removes the peg from the hole 2 (Step 3) and places it into the hole 4 -- and finds the optimum Golomb ruler of length 6 with 4 pegs.*

(b) *Decisions about which peg to move from one hole to another hole can be gleaned from the data structure defined as the Hasse graph: the label of each vertex is a solution defined by the pair* **coordinate:value**

Depending on puzzle's initial coordinate, walkLengths can vary from 0 to 6 steps, with a median of 1, and a mean of 1.9.

Figure 2: Two representations of a small Golomb ruler puzzle ogr(m,n) for m = 4, n = 6: (a) an arrangement of pegs simulated by a tablet device, (b) a complete Hasse graph as a data structure to guide decisions when solving the puzzle under a self-avoiding walk.

get solution is not found, reverse the walk until a new coordinate is reached at rank 1, then reverse the walk towards the best coordinate at rank $m_{max} - 2$ again, etc.

3 *Notation and Definitions*

We keep the notation and definitions simple, leveraging the illustrative example in Figure 2. A combinatorial problem is defined by its (objective) *function* and its *coordinates*. Coordinates are represented as a set of strings, such as 01011... for a binary coordinate of type B, 210210... for a ternary coordinate of type T, $4, 2, 5, 3, \ldots$ for a permutation coordinate of type P, etc. We can also *concatenate* coordinates of different types. For example, we define the 2D protein folding problem *on a square lattice* by computing its function values with coordinates represented as *a concatenation of binary and ternary coordinates*. We say that the function $\Theta(\underline{\varsigma})$ returns a *value* associated with a specific *coordinate* ς. Frequently, we denote *coordinate length* or *coordinate dimension* with the symbol L or n.

A DATA STRUCTURE of special importance for any combinatorial problem or puzzle is the *Hasse graph*, an extension of the well-known *Hasse digram*[15]. Each vertex is assigned a label $\underline{\varsigma} : \Theta(\underline{\varsigma})$ where $\underline{\varsigma}$ represents a unique n-dimensional coordinate string $\underline{\varsigma}$, $\Theta(\underline{\varsigma})$ is the value of the function Θ for $\underline{\varsigma}$. Interchageably, we associate each vertex in the Hasse graph with a face of a multifaceted hyper dice or hyperhedron and vertex adjacencies in the graph with *the face adjacencies* in the hyperhedron. In this context *we do not* associate a regular hexahedron (a cube) that also has 8 vertices with a hypercube graph of $2^3 = 8$ vertices. Rather, we associate the 6 *faces* of the hexahedron with $3! = 6$ *permutation coordinates* of length 3 with vertices in the Hasse graph, and we associate the 8 faces of octahedron with $2^3 = 8$ *binary coordinates* of length 3 with vertices in another Hasse graph. The data structure of *any* Hasse graph makes it simple to define:

coordinate rank: Under k-ary coordinates ($k \geq 2$), the rank of each coordinate is the sum of all values in the coordinate string. For example, the rank of a binary coordinate `101011` is 4; the rank of a ternary coordinate `201021` is 6. There is only one coordinate with a maximum rank: for a binary coordinate of length $L = 6$, the maximum rank is 6 and the the coordinate `111111` is the topmost coordinate in the Hasse graph. For a ternary coordinate of length $L = 6$, the maximum rank is 12 and the the coordinate `222222` is the topmost coordinate in the Hasse graph. On the opposite end of the Hasse graph is the coordinate with rank 0, alway a string of all 0's. For an example of concatenated binary/ternary coordinates, see Figure 10 in the Appendix.

With permutation coordinates, the rank of a permutation coordinate is the inversion number of the permutation. For example, the rank of the coordinate `4,2,5,1,6,3` is 7. For a permutation coordinate of length $L = 6$, the maximum rank is $(6 \times 5)/2 = 15$ and the the coordinate `6,5,4,3,2,1` is the topmost coordinate in the Hasse graph. On the opposite end of the Hasse graph is the coordinate with rank 0, a permutation in the natural order, `1,2,3,4,5,6`.

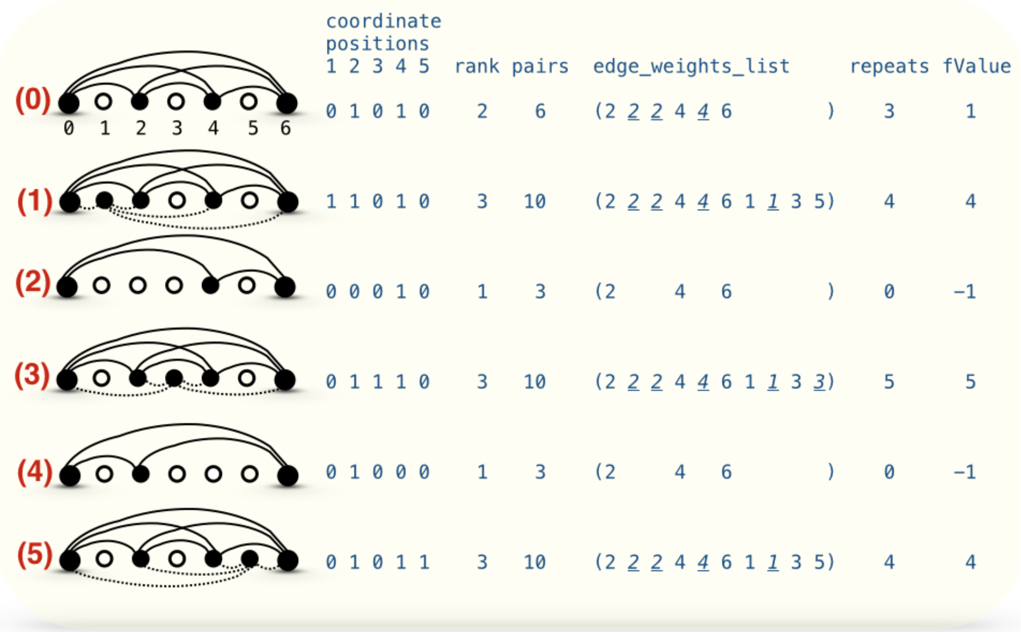
coordinate distance: Coordinate distance is defined for a pair of coordinates. It is computed as the absolute difference of two coordinate ranks.

coordinate pivot: Coordinate pivot is associated with a step during the walk. For step = 0, the pivot coordinate is usually called the initial coordinate. The pivot coordinate is also the reference coordinate for the set of neighborhood coordinates.

coordinate neighborhood: The default definition of coordinate neighborhood is the set of all coordinates with distance of 1 from the pivot coordinate. For binary coordinates of length L, the size of coordinate neighborhood under the distance of 1 is always L. For permutation coordinates of length L, the size of coordinate neighborhood under the distance of 1 is always $L - 1$. However, for k-ary coordinates ($k > 2$, the size of coordinate neighborhood under the distance of 1 is *not* constant, it depends on the rank of the coordinate. See Figure 9b and Figure 10 as an example where neighborhood size is not constant.

neighborhood probing under the 'simple' option: The word *probe* refers to a single

[15] `http://en.wikipedia.org/wiki/Hasse_diagram`



	coordinate positions 1 2 3 4 5	rank	pairs	edge_weights_list	repeats	fValue
(0)	0 1 0 1 0	2	6	(2 $\underline{2}$ $\underline{2}$ 4 $\underline{4}$ 6)	3	1
(1)	1 1 0 1 0	3	10	(2 $\underline{2}$ $\underline{2}$ 4 $\underline{4}$ 6 1 $\underline{1}$ 3 5)	4	4
(2)	0 0 0 1 0	1	3	(2 4 6)	0	-1
(3)	0 1 1 1 0	3	10	(2 $\underline{2}$ $\underline{2}$ 4 $\underline{4}$ 6 1 $\underline{1}$ 3 $\underline{3}$)	5	5
(4)	0 1 0 0 0	1	3	(2 4 6)	0	-1
(5)	0 1 0 1 1	3	10	(2 $\underline{2}$ $\underline{2}$ 4 $\underline{4}$ 6 1 $\underline{1}$ 3 5)	4	4

Figure 3: The tableau in this figure relates to the initial positions of pegs in Figure 2. The four pegs define a complete graph at line (0) and the corresponding pivot coordinates `01010`; the `1's` represent all pegs except the pegs at the fixed positions of `0` and `6`. For each edge at positions i, j, we assign a weight $|i - j|$. The five configuration of graphs, (1)-(5), represent the adjacent coordinates of the pivot coordinate and are created by removing a vertex from the complete graph or adding a vertex to the complete graph.

Under *simple probing*, we evaluate the objective function explicitly each time we visit the adjacent coordinate. However, under *tableau probing*, we only need to evaluate the effect of adding or removing the peg from the puzzle, re-using the information computed when evaluating the pivot coordinate in the *edgeWeightList* shown in the tableau. The weights that repeat in this list are shown in in *italics* and are also underlined.

Under the tableau method and with good data structures, the relative runtime of probing the entire neighborhood decreases rapidly as the size of the neighborhood increases with the problem size.

evaluation of the objective function $\Theta(\underline{\varsigma})$ for a given coordinate $\underline{\varsigma}$. In any combinatorial solver, the total number of such probes correlates with the solver runtime and is independent of the computational platform. The term *neighborhood probing* under the option 'simple' refers to explicit evaluation of neighborhood function values under the assumption that all coordinates being probed are independent. For the neighborhood of size L, the total cost of neighborhood probing under the option 'simple' is therefore $O(L)$.

neighborhood probing under the 'tableau' option: The neighborhood probing under the 'tableau' option implies that we have worked out, specifically for each problem, the objective function dependencies when probing of *all* neighborhood coordinates with respect to a single coordinate: the pivot coordinate. With increasing value of L, the cost of probing the entire the neighborhood under the 'tableau' option is significantly less than $O(L)$, see the results in Section 4. Implementation details of the 'tableau' option are specific for each objective function and are a new feature in all of our articles on self-avoiding walk search after 2013. For an illustration that guides the implementation of the tableau for the `ogr` solver, see Figure 3.

contiguous and non-contiguous walk segment: Let the coordinate $\underline{\varsigma}_0$ be the initial coordinate from which the walk takes the first step. Then the sequence walkList$_\omega = \{\underline{\varsigma}_0, \underline{\varsigma}_1, \underline{\varsigma}_2, \ldots, \underline{\varsigma}_j, \ldots, \underline{\varsigma}_\omega\}$ is called a *walk segment* of length ω, the coordinates $\underline{\varsigma}_j$ are denoted as *pivot coordinates* and $\Theta(\underline{\varsigma}_j)$ are denoted as *pivot values*. Assuming that the walk has been instrumented to search for a minimum of $\Theta(\underline{\varsigma})$ and that Θ^{ub} denotes the best upper bound, we say that the walk *reaches* its target value (and stops) when $\Theta(\underline{\varsigma}_\omega) \leq \Theta^{ub}$.

We denote d_j as the rank distance between two adjacent pivot coordinates $(\underline{\varsigma}_j, \underline{\varsigma}_{j-1})$: $d_j = \text{rankDistance}(\underline{\varsigma}_j, \underline{\varsigma}_{j-1}) \quad \forall j \in 1, 2, ..., \omega$ and say that the walk segment is *contiguous* if $d_j = 1 \quad \forall j \in 1, 2, ..., \omega$ and *non-contiguous* otherwise.

```
Search procedure based on a Self-Avoiding Walk (SAW)
procedure SAW.wander()
    while true do
        (a) select a random coordinate and mark it as the 'initial pivot';
        (b) probe all unmarked adjacent coordinates, then select and mark
            the coordinate with the 'best value' as the new pivot;
        (c) continue the walk until either the 'best value' $\leq$ 'target value' or
            the walk is being blocked by adjacent coordinates that are already
            pivots;
        (d) if the walk is blocked, restart the walk from a randomly selected
            'new initial pivot';
        (e) manage the memory constraints with an efficient data structure
            such as a hash table.
    end while
end procedure
```

Figure 4: A *walk* as a sequence of steps that chain a set of *pivot coordinates*, *adjacent coordinates* as the *local neighborhood* of the pivot coordinate, and *probing* as evaluating the function for values in the local neighborhood. A *feasible solution* of the combinatorial problem is a pair *(coordinatePivot*:*valuePivot)*.

Under *this* simple strategy, named as a walk under the option 'wander' the issue of self-avoiding walks getting trapped during the search has been a rare event, observed on instances with relatively small coordinate dimension only – i.e. the target value is almost always found first.

objective function: Our formulation of the objective function for the ogr puzzle is in terms of *number of repeats in ruler lengths* and the *rank* of the coordinate.

$$\Theta(\underline{\varsigma}) = \begin{cases} \text{numRepeats}, & \text{if } rank > m_{max} - 1. \\ \text{numRepeats} - \text{rank}, & \text{otherwise.} \end{cases}$$

Under this formulation, the coordinate $\underline{\varsigma}^*$ returns the global minimum of the objective function (or the solution of the puzzle) with the value of

$$\Theta(\underline{\varsigma}^*) = -m_{max} + 2$$

self-avoding walk segment: We say that the walk segment is *self-avoiding* if all pivots in the walkList$_\omega$ are unique. We say that the walk is composed of two or more *walk segments* if the initial pivot of each walk segment has been induced by a well-defined heuristic such as *random restarts*. Walk segments can be of different lengths and if viewed independently of other walks, may be self-avoiding or not. A walk composed of two or more self-avoiding walk segments may no longer be a self-avoiding walk, since some of the pivots may overlap and also form cycles. Also, a walk composed of two or more self-avoiding walk segments is almost certainly no longer a contiguous walk.

self-avoding search algorithm: See Figure 4 for a description of the version of the self-avoding search algorithm (option 'wander').

4 Summary of Experiments

For a summary of experiments, see Figures 5, 6 and 7. To rapidly explore the most promising self-avoding walk strategies, we prototyped the solver in a scripting language tcl. The asymptotic performance of the various solver options summarized in this section will be replicated with a faster implementation in C++ so we can explore the solution space for larger instances. What is most important in this phase is to measure the asymptotic solver performance not in terms

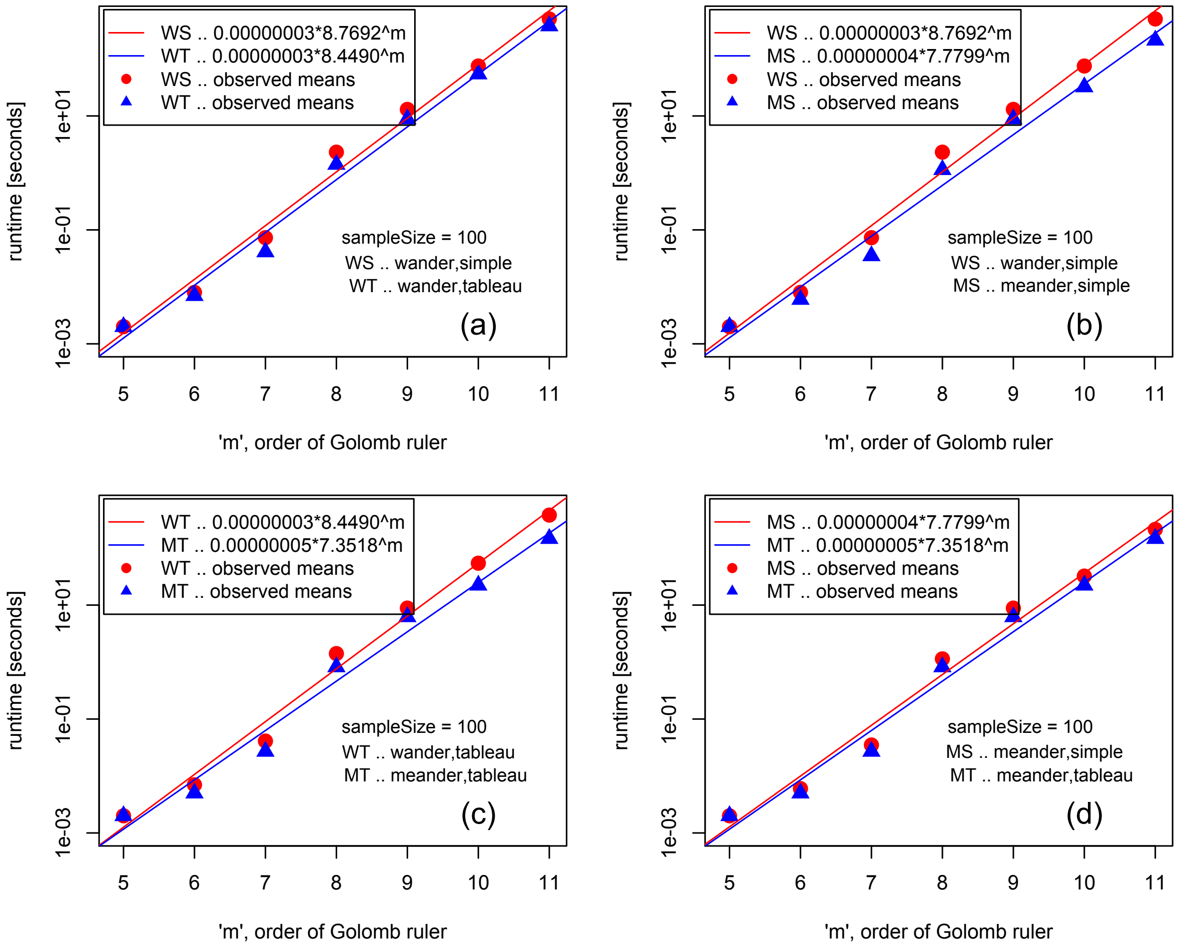


Figure 5: In these experiments we summarize the average case runtime performance of the `ogr` solver under four configurations: `WS-WT`, `WS-MS`, `WT-MT`, and `MS,MT`. The best solver is clearly the solver with option of `MT` (meander-tableau) which is not entirely unexpected, considering the hints we can get by observing the Hasse graph in Figure 2. The asymptotic model for the `MT` solver runtime mean (in seconds) is thus $0.00000005 \ast 7.3518^m$

of runtime alone; the total count of probes is independent of computational platform as is the number of steps taken by the walk. By measuring these variables *and* runtime we could explain why solvers that rely on self-avoding walk strategies and the probing based on the 'tableau' option outperform the currently available subset of alternative solver on number of hard combinatorial problems, including the `ogr` problem.

All experiments are summarized in terms of four solver options:

`wander-simple (WS): wonder` implies the walk as described in Figure 4; under `simple` we probe neigborhood at the cost of $O(L)$.

`wander-tableau (WT:)` Under `tableau` we probe neigborhood of size L at the cost much less than $O(L)$ as L increases.

`meander-simple (MS): meander` implies the walk as illustrated in the Hass graph in Figure 2, steps are always taken from rank of $m_{max} - 2$ to rank $m_{max} - 3$ and back, until the optimum coordinate

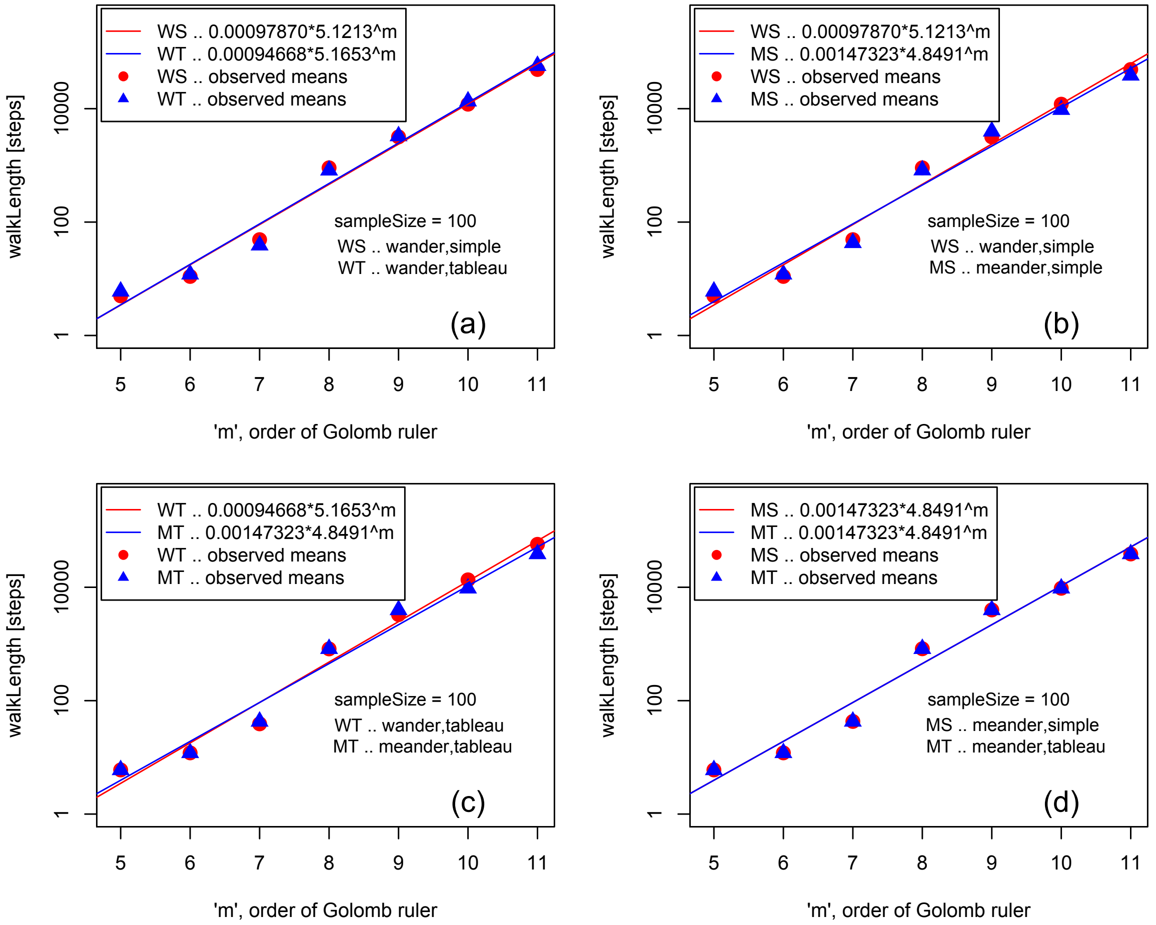


Figure 6: These experiments are in 1-to-1 correspondence with the runtime experiments in Figure 6: here we measure the walkLenth instead or runtime. The walkLengths returned by WS and WT are very close, and most remarkably, walkLengths returned by MS and MT are equivalent. There is an explanation for this in Figure 7.

The asymptotic walkLengths models such as $0.00147323 * 4.8491^m$ will help us predict not only the average requirements for size of the hash table that stores the pivot coordinates of the walk segment, it will also predict the maximum requirement since we know that the runtime as well as walkLength have exponential distrubution, implying that the value of the mean is approximately the value of the standard deviation.

is found at rank $m_{max} - 2$.

meander-tableau (MT): meander and tableau, described above.

The plots in Figures 5, 6 and 7 bring together results in pairs of either WS-WT, WS-MS, WT-MT, and MS-MT. From Figure 5 we learn that the best solver is the solver with option of MT (meander-tableau) – which should be implemented in C++ first. The asymptotic model for the MT solver runtime mean (in seconds) is $0.00000005 * 7.3518^m$.

From Figure 6 we learn that while there are significant runtime differences between the four solver options, there are only small or no walkLength differences between these solvers. The asymptotic walkLength models will help in predicting the size requirements for the hash tables. The correlations in Figure 7 demonstrate the consistency of the solver implementations and good practices in designing these experiments.

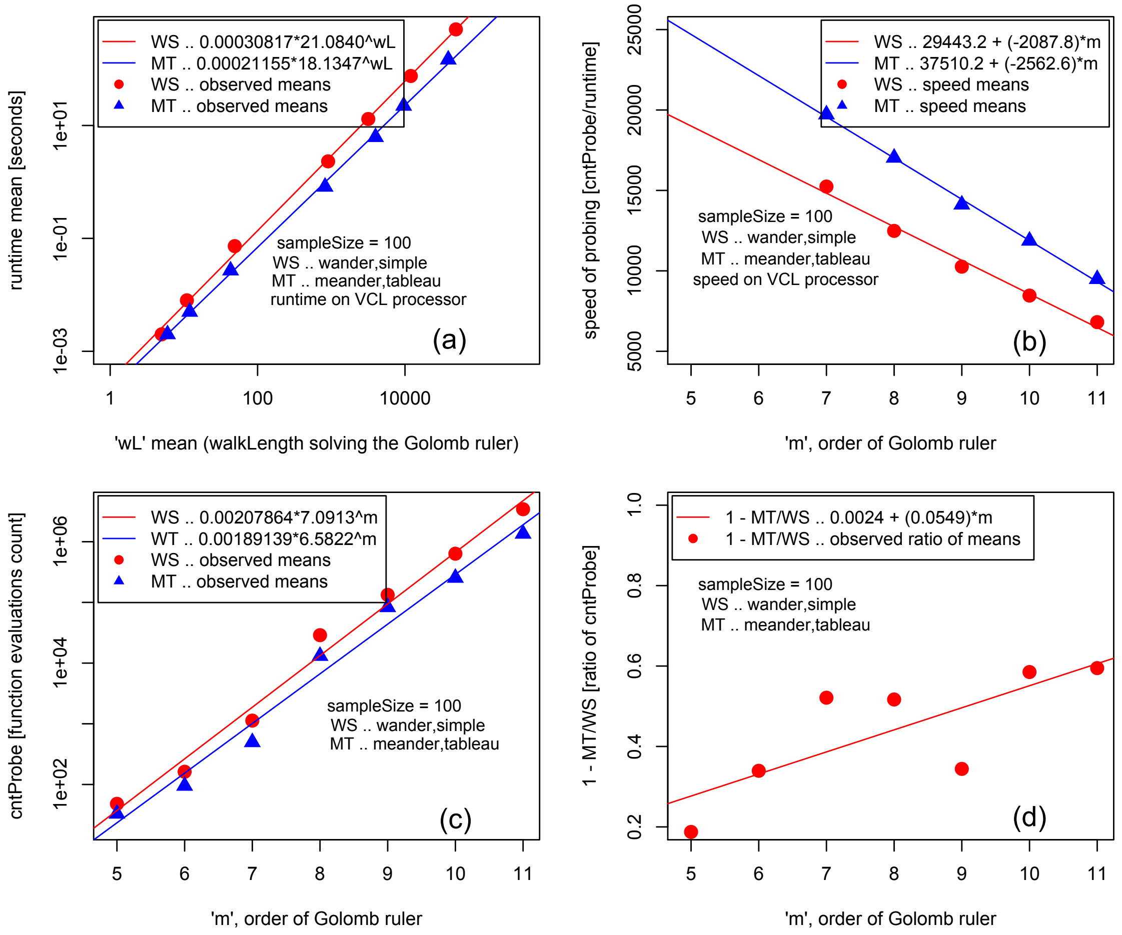


Figure 7: In this figure, we contrast a number of observed variables. In (a), we not only observe two sets of perfect correlations between the walkLength mean and runtime mean, we also observe that the runtime mean under the option MT is consistently and significantly better than the runtime mean under the option WS. In (b), we confirm again that the runtime speed, measured with ratio of runtime to cntProbe is much higher under the option MT. The plots of cntProbe in (c) are clearly correlated with respective runtime plots in Figure 5. In (d), we observe a positive slope when computing the 1 - ratio of cntProbe values for MT and WS.

5 Conclusions

The asymptotic model for the MT solver runtime mean (in seconds) is $0.00000005 * 7.3518^m$. Under the C++ implementation, we expect the runtime to reduce by a factor of about 1000. Thus, we can predict the mean runtime for C++ solver as $O(0.00000000005 * 7.3518^m)$. When we run this solver for $m \in (16, 17, 18, 19, 20)$ we can expect, since we have an exponential runtime distribution, a solution hit rate of 63% in $(1.01, 7.4, 54.6, 402, 2955)$ hours. None of the existing memetic/tabu/localSearch solvers report comparable results for even $m = 16$.

The Appendix outlines some of the instrumentation techniques and the xBed infrastructure we used to design, execute, and evaluate the experiments summarized in this paper.

A Appendix: About xBed

This appendix highlights the xBed infrastructure and instrumentation techniques used to design, execute, and evaluate our experiments:

xBed schema in Figure 8 on GitHub server http://github.com/fbrglez/gitBed/.

Door-key tablet puzzles in Figure 9 described in the *The Fable About Urns*.

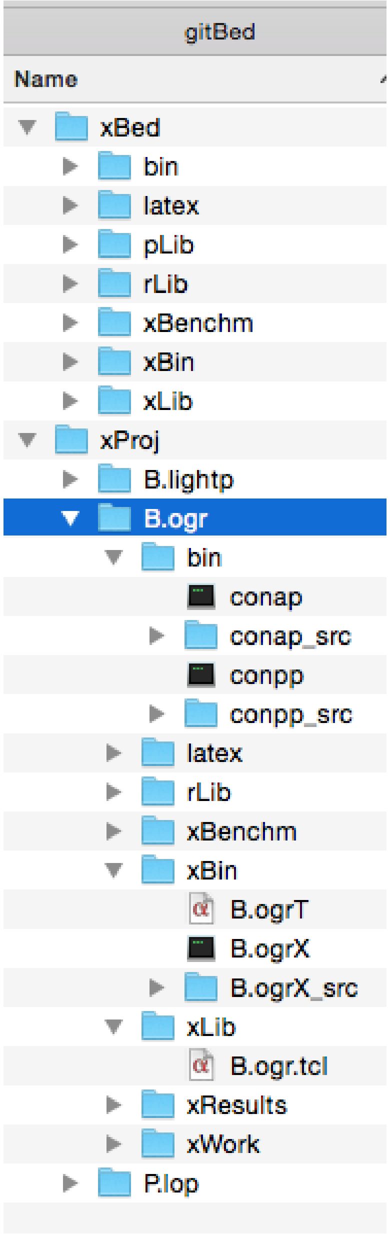
Walks in Hasse graphs in Figure 10 described in the *The Fable About Urns*.

Standardized solver I/O in Figure 11 described in the *Solver Standardization via Encapsulation*.

Asymptotic performance experiments in Figure 12 described in the *Asymptotic Performance Experiments*.

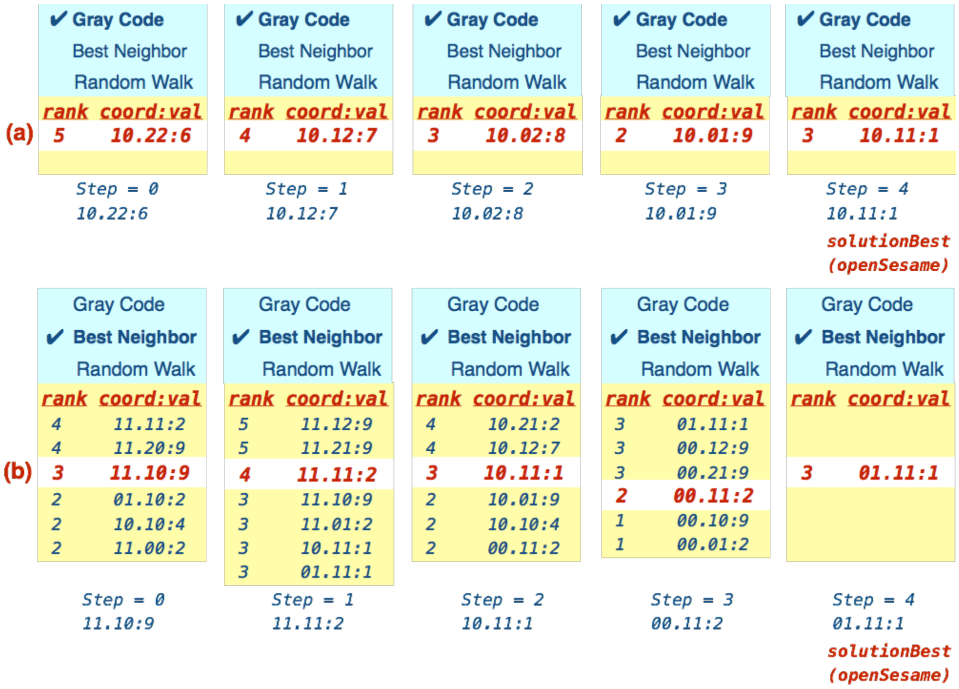
THE xBed SCHEMA in Figure 8 has been set up on GitHub server http://github.com/fbrglez/gitBed/ in 2015 to support a project-oriented class on Combinatorial Optimization and to facilitate collaboration between distributed participants. The repositories under xBed are software libraries, software executables, and data sets which are shared by all participants. The repositories under xProj are software libraries, software executables, and data sets that are *problem-specific* are called *sandboxes*. Under xResults, each sandbox archives summaries of computational experiments for publications. Each experiment *is reproducible by design*, i.e. the initial seeds for each experiment are included in the archive.

THE FABLE ABOUT URNS involves three student friends: Fred from the School of Statistics, Gretel from the School of Computer Science, and Hansel from the School of Geomatics. Fred is a practical joker motivated by teaching others about problem solving, experiments, and statistics. Gretel and and Hansel live in adjacent apartments. One evening Fred replaces entry door locks of each apartment with a card reader and places under each door 100 urns with labels [001,..,100] and $n = 2^2 \times 3^3 = 36$ uniquely numbered small cards in each urn. Each card has an 8-digit number uuu.xx.yy:z where uuu denotes the number of the urn, xx is a pair of binary digits, yy is a pair of ternary digits, and $z = \Theta(\text{xx.yy})$. We say that xx.yy is the coordinate of the function $\Theta(.)$ and that z denotes the value of this function for the given coordinate. The strings following the urn number, xx.yy:z, are the same for each urn. Only one key or string in *xx.yy:z* opens the door of Gretel's apartment, only one string in *xx.yy:z*, different from Gretel's key, opens the door of Hansel's apartment. Once the card is inserted, the lock decodes the string and turns the door light to red if the card does not open the door and turns the light to green if the card *could* open the door, thereby signaling that another urn should be started for the experiment. The door light



```
▼ xBed
    ▶ bin
    ▶ latex
    ▶ pLib
    ▶ rLib
    ▶ xBenchm
    ▶ xBin
    ▶ xLib
▼ xProj
    ▶ B.lightp
    ▼ B.ogr
        ▼ bin
              conap
            ▶ conap_src
              conpp
            ▶ conpp_src
        ▶ latex
        ▶ rLib
        ▶ xBenchm
        ▼ xBin
              B.ogrT
              B.ogrX
            ▶ B.ogrX_src
        ▼ xLib
              B.ogr.tcl
        ▶ xResults
        ▶ xWork
    ▶ P.lop
```

Figure 8: The xBed schema and examples of three collaborative xBed projects (or sandboxes): B.lightp, B.ogr, and P.lop. The coordinate type of B.lightp is B, a binary string, and the objective function lightp defines the lights-out puzzle. The coordinate type of B.ogr is B, a binary string, an the objective function ogr defines the optimal Golomb ruler problem. The coordinate type of P.lop is P, a permutation string, and the objective function lop defines the linear ordering problem.



(a)

rank	coord:val
5	10.22:6

Step = 0
10.22:6

rank	coord:val
4	10.12:7

Step = 1
10.12:7

rank	coord:val
3	10.02:8

Step = 2
10.02:8

rank	coord:val
2	10.01:9

Step = 3
10.01:9

rank	coord:val
3	10.11:1

Step = 4
10.11:1
solutionBest
(openSesame)

(b)

rank	coord:val
4	11.11:2
4	11.20:9
3	11.10:9
2	01.10:2
2	10.10:4
2	11.00:2

Step = 0
11.10:9

rank	coord:val
5	11.12:9
5	11.21:9
4	11.11:2
3	11.10:9
3	11.01:2
3	10.11:1
3	01.11:1

Step = 1
11.11:2

rank	coord:val
4	10.21:2
4	10.12:7
3	10.11:1
2	10.01:9
2	10.10:4
2	00.11:2

Step = 2
10.11:1

rank	coord:val
3	01.11:1
3	00.12:9
3	00.21:9
2	00.11:2
1	00.10:9
1	00.01:2

Step = 3
00.11:2

rank	coord:val
3	01.11:1

Step = 4
01.11:1
solutionBest
(openSesame)

Figure 9: Two views of the door-key tablet puzzles under the binary/ternary (BT) coordinate system: (a) by default, the tablet next to Gretel's entrance expects Gretel to solve the puzzle using the *Gray Code* option, (b) by default, the tablet next to Hansel's entrance expects Hansel to solve the *Best Neighbor* option under the self-avoiding walk.

turns to flashing green and the door actually opens when the experiment is completed successfully with the 100-th urn.

In the evening, Gretel and Hansel return to their apartments from a party. Who gets in first? Watching Gretel, she takes the card from the urn and if she does not succeed in opening the door, she puts the card into her handbag and retrieves another card from the urn. Hansel, who had a few drinks at the party, takes the card and if he does not succeed in opening the door, returns the card to the urn. We say that Gretel is sampling contents of the urn without replacement, while Hansel is sampling with replacement. The probability of Gretel finding the correct card on trial k follows uniform distribution, given n cards: the probability is $1/n$, the mean value is $(n+1)/2$, and the variance is $(n^2-1)/12$. However, the probability of Hansel finding the correct card on trial k follows geometric distribution: the probability is $(1/n)(1-(1/n))^{k-1}$, the mean value is n, and the variance is $n^2(1-(1/n))$. The point of the fable so far: we learn that in search scenarios such as described here, one can improve the chance of *first success* by dynamically reducing the search space after each trial.

Replacing urns with door-key tablet puzzles. The next evening, Fred replaces the urns in front of each door with two tablet computers, each cemented to the wall next to the door of each apartment. The door lock is invisible from the outside, it wirelessly takes commands from the adjacent tablet. For initial tablet display as seen by

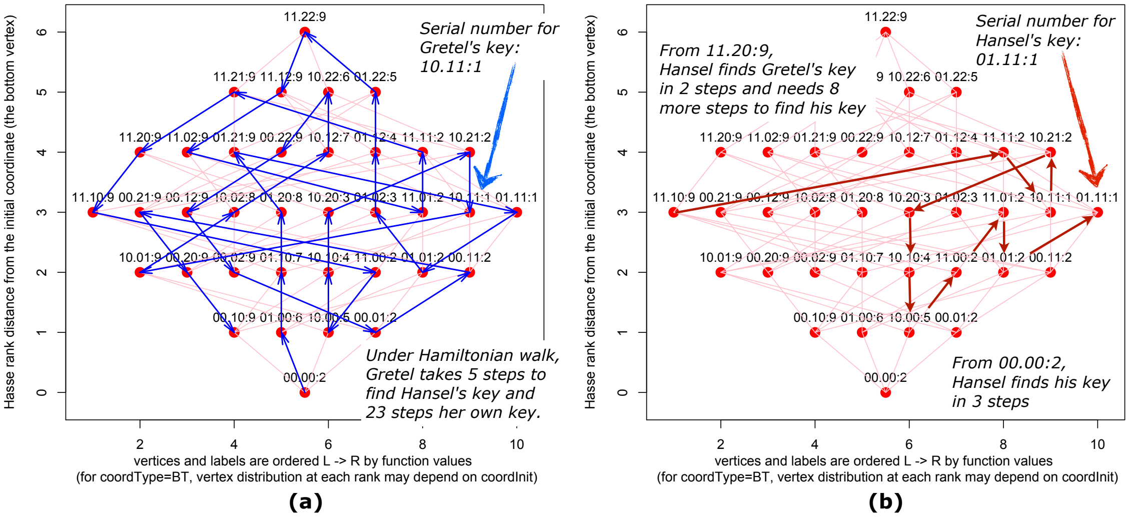

Hasse rank distance from the initial coordinate (the bottom vertex)
0
1
2
3
4
5
6
11.22:9
11.21:9 11.12:9 10.22:6 01.22:5
11.20:9 11.02:9 01.21:9 00.22:9 10.12:7 01.12:4 11.11:2 10.21:2
11.10:9 00.21:9 00.12:9 10.02:8 01.20:8 10.20:3 01.02:3 11.01:2 10.11:1 01.11:1
10.01:9 00.20:9 00.02:9 01.10:7 10.10:4 11.00:2 01.01:2 00.11:2
00.10:9 01.00:6 10.00:5 00.01:2
00.00:2
Serial number for Gretel's key: 10.11:1
Under Hamiltonian walk, Gretel takes 5 steps to find Hansel's key and 23 steps her own key.
2
4
6
8
10
vertices and labels are ordered L -> R by function values
(for coordType=BT, vertex distribution at each rank may depend on coordInit)
(a)
From 11.20:9, Hansel finds Gretel's key in 2 steps and needs 8 more steps to find his key
Serial number for Hansel's key: 01.11:1
9 10.22:6 01.22:5
From 00.00:2, Hansel finds his key in 3 steps
(b)

Figure 10: Walks in two identical Hasse graphs as data structures for the door-key tablet puzzles under the binary/ternary coordinate systems: (a) for each step, Gretel is chooses the next coordinate under the Gray code rule and completes a Hamiltonian walk on the graph while also finding the key to her apartment on step 23, (b) for each step, Hansel chooses the ajacent (neighborhood) coordinate with the best value and finds his apartment key, under the self-avoiding walk strategy, on step 10.

For the average average walk length we can show (1) random walk strategy is equivalent to sampling the urn without replacement ($2^2 \times 3^3$), (2) the walk taken by Gretel is at $O(0.5 \times 2^2 \times 3^3)$ an improvement over the random walk, and (3) the walk taken by Hansel is at $O(0.25 \times 2^2 \times 3^3)$ an improvement over the walk by Gretel.

Gretel see Figure 9a above the line `Step = 0`. For initial tablet display as seen by Hansel see Figure 9b above the line `Step = 0`.

Since Gretel is in majoring in Computer Science, Fred pre-selects *Gray Code* strategy for her walk; he asssumes that Gretel is familiar with binary/ternary Gray codes and will select each coordinate for the walk such that rank distance between the codes is always 1. The rank of each concatenated binary/ternary coordinate is determined by the sum of coordinate component values, e.g. `1+0+2+2 = 5`.

Since Hansel is majoring in Geomatics, Fred pre-selects *Best Neighbor* strategy for the walk (the tablet maintains the self-avoiding walk principles under this strategy) and assumes that Hansel will exploit the information about *function values* associated with each *neighborhood coordinate*: the rank of these coordinates is ± 1 the rank of the reference coordinate. Hansel makes the decision about the next step consistently: he chooses the coordinate with the lowest function value; if there is more than 1 such coordinate, the choice is random.

Decisions made by Gretel and Hansel in Figure 9 demonstrate that both can find keys that open their apartments in 4 steps. However, with tablets enforcing the rule to repeat the experiment 100 times (as Fred has enforced it with 100 urns) we find that, in terms of average walk length, the strategy of *Best Neighbor* outperforms the strategy of *Gray Code* and that the strategy of *Random Walk* is equivalent to sampling the urns with replacement.

The Hasse graphs in Figure 10 are the implicit data structures that help us not only to formulate the coordinate ranks and coordinate neighborhoods, they also serve to illustrate many types of walks: from random walks that typically will have a number of cycles, to

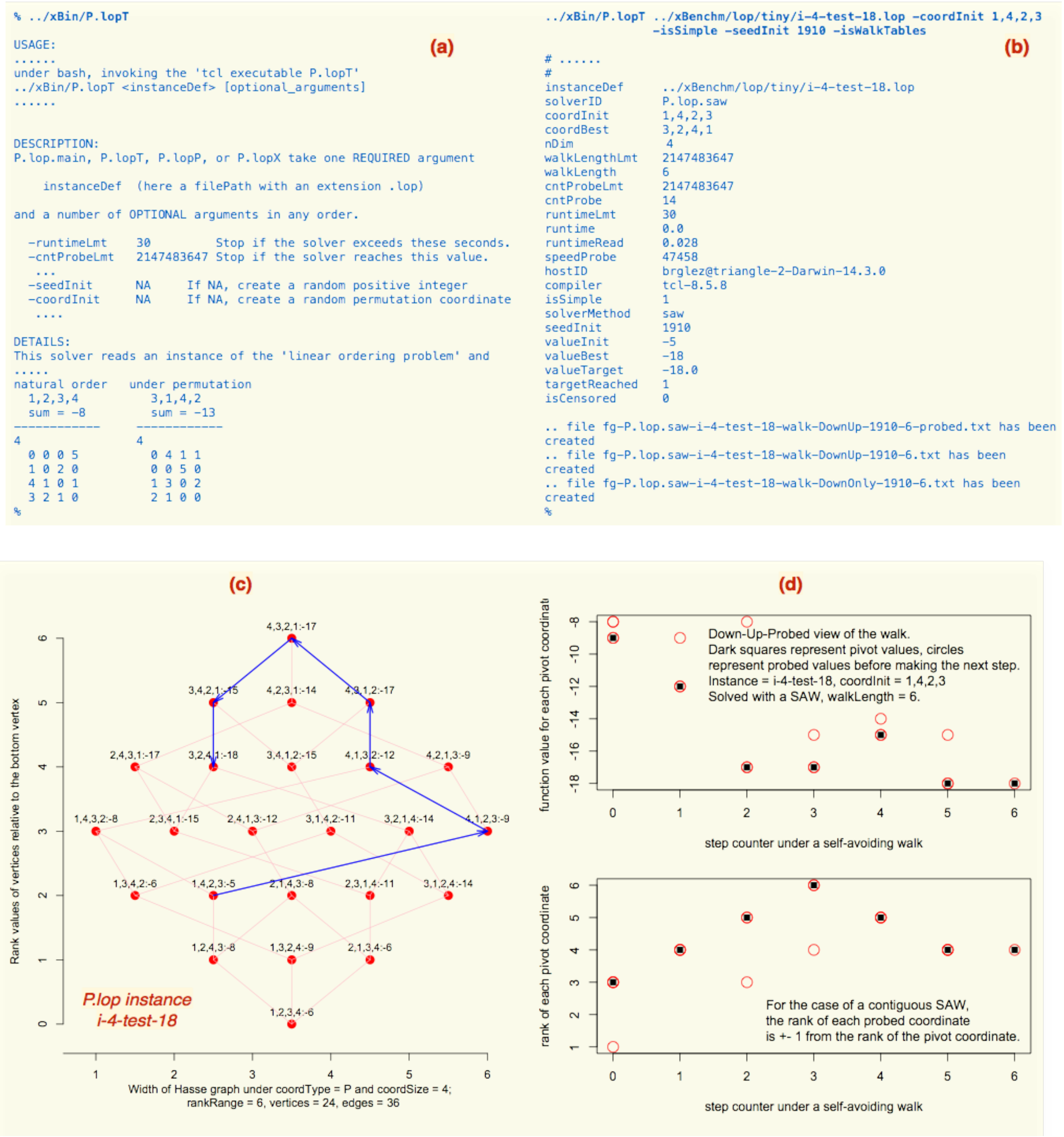
Hamiltonian walks in Figure 9a which are self-avoiding by definition, and also to relatively long contiguous self-avoiding walk segments such as illustrated in Figure 9b. Vertices traversed from one coordinate rank to another during the walk are in two categories: (1) only binary coordinate is changing, the ternary coordinate is fixed; (2) the binary coordinate is fixed, only ternary coordinate is changing. The *rank* of each coordinate is defined as the arithmetic sum of coordinate numbers in each coordinate string: for example, the rank of coordinates `11.01` and `00.21` is 3.

SOLVER STANDARDIZATION VIA ENCAPSULATION is a standard scripting technique to modify the solver I/O. It is rarely the case that command lines for a collection of solvers, designed by different teams, will have similar command lines and similar standard output formats after solver invocation. The solver encapsulation under the `xBed` environment makes the solver command line compatible with the syntax of other solvers under test. After such encapsulation, the task of solver invocation for a large number of performance experiments is not only greatly simplified, it is also standardized. The solver encapsulation also supports the standardization of the solver output formats, thereby making performance evaluation and comparisons not only fair but also statistically significant.

An example of such solver standardization is shown in Figures 11a and 11b. The Figure 11a illustrates a snippet of a *standardized command line query* for the solver `P.lopT` that resides under `gitBed/xProj/P.lop/xBin` as illustrated in the schema in Figure 8. The Figure 11b illustrates a snippet of a a *standardized solver command line invocation* and a *standardized solver output*. Given this standardization of solver IO, it is now relatively simple to design and execute a large number comparative performance experiments that are not only statisticaly significant but also reproducible since the inital seeds for each invocation have been archived.

Examples of statistical summaries of such experiments are illustrated in Figure 12.

ASYMPTOTIC PERFORMANCE EXPERIMENTS are one of the best way to evaluate the performance differences of two solvers. Given a set of problem instances of increasing size and hardness, we repeatedly invoke, under different initial seeds, the solvers under standardized `xBed` encapsulation and tabulate runtimes, walk lengths, etc for each seed and each solver. Examples of statistics collected for such experiments are tablated in Figure 12c. The three tables summarize experimental results for the linear ordering problem solver `P.lopT`, reporting performance statistics such as *median*, *mean*, *standard devi-*


(a)
(b)
(c)
4,3,2,1:-17
3,4,2,1:-15
4,2,3,1:-14
4,3,1,2:-17
2,4,3,1:-17
3,2,4,1:-18
3,4,1,2:-15
4,1,3,2:-12
4,2,1,3:-9
1,4,3,2:-8
2,3,4,1:-15
2,4,1,3:-12
3,1,4,2:-11
3,2,1,4:-14
4,1,2,3:-9
1,3,4,2:-6
1,4,2,3:-5
2,1,4,3:-8
2,3,1,4:-11
3,1,2,4:-14
1,2,4,3:-8
1,3,2,4:-9
2,1,3,4:-6
1,2,3,4:-6
P.lop instance
i-4-test-18
Rank values of vertices relative to the bottom vertex
0
1
2
3
4
5
6
1
2
3
4
5
6
Width of Hasse graph under coordType = P and coordSize = 4;
rankRange = 6, vertices = 24, edges = 36
(d)
function value for each pivot coordinate
-8
-10
-12
-14
-16
-18
Down-Up-Probed view of the walk.
Dark squares represent pivot values, circles
represent probed values before making the next step.
Instance = i-4-test-18, coordInit = 1,4,2,3
Solved with a SAW, walkLength = 6.
0
1
2
3
4
5
6
step counter under a self-avoiding walk
rank of each pivot coordinate
1
2
3
4
5
6
For the case of a contiguous SAW,
the rank of each probed coordinate
is +- 1 from the rank of the pivot coordinate.
0
1
2
3
4
5
6
step counter under a self-avoiding walk

```
% ../xBin/P.lopT

USAGE:
......
under bash, invoking the 'tcl executable P.lopT'
../xBin/P.lopT <instanceDef> [optional_arguments]
......


DESCRIPTION:
P.lop.main, P.lopT, P.lopP, or P.lopX take one REQUIRED argument

    instanceDef  (here a filePath with an extension .lop)

and a number of OPTIONAL arguments in any order.

  -runtimeLmt    30         Stop if the solver exceeds these seconds.
  -cntProbeLmt   2147483647 Stop if the solver reaches this value.
   ...
  -seedInit      NA      If NA, create a random positive integer
  -coordInit     NA      If NA, create a random permutation coordinate
   ....

DETAILS:
This solver reads an instance of the 'linear ordering problem' and
.....
natural order   under permutation
  1,2,3,4         3,1,4,2
  sum = -8        sum = -13
------------    ------------
4               4
  0 0 0 5         0 4 1 1
  1 0 2 0         0 0 5 0
  4 1 0 1         1 3 0 2
  3 2 1 0         2 1 0 0
%
```

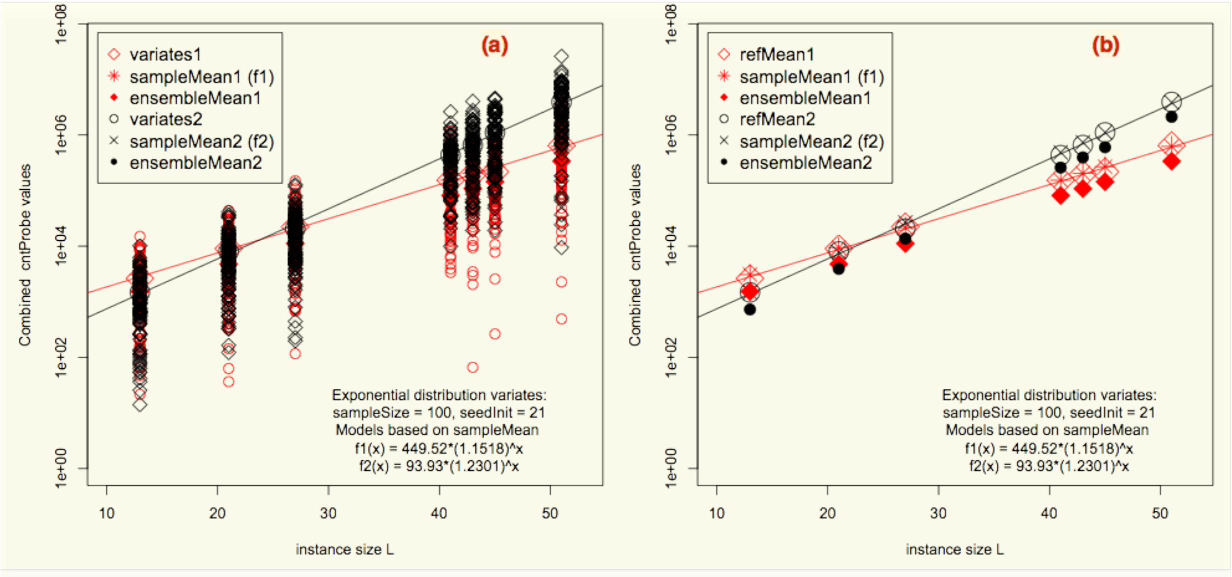
```
../xBin/P.lopT ../xBenchm/lop/tiny/i-4-test-18.lop -coordInit 1,4,2,3
               -isSimple -seedInit 1910 -isWalkTables

# ......
#
instanceDef     ../xBenchm/lop/tiny/i-4-test-18.lop
solverID        P.lop.saw
coordInit       1,4,2,3
coordBest       3,2,4,1
nDim             4
walkLengthLmt   2147483647
walkLength      6
cntProbeLmt     2147483647
cntProbe        14
runtimeLmt      30
runtime         0.0
runtimeRead     0.028
speedProbe      47458
hostID          brglez@triangle-2-Darwin-14.3.0
compiler        tcl-8.5.8
isSimple        1
solverMethod    saw
seedInit        1910
valueInit       -5
valueBest       -18
valueTarget     -18.0
targetReached   1
isCensored      0

.. file fg-P.lop.saw-i-4-test-18-walk-DownUp-1910-6-probed.txt has been
created
.. file fg-P.lop.saw-i-4-test-18-walk-DownUp-1910-6.txt has been
created
.. file fg-P.lop.saw-i-4-test-18-walk-DownOnly-1910-6.txt has been
created
%
```

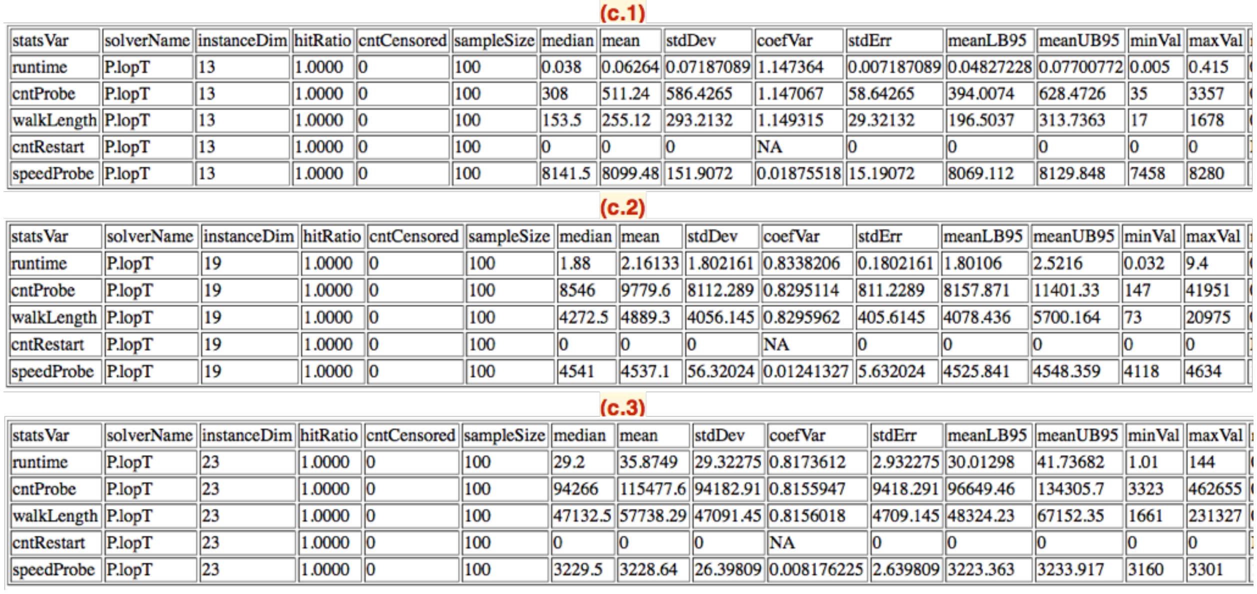
Figure 11: Standardized solver I/O: (a) command line query, (b) standard output, (c) a walk in a Hasse graph, (d) stepwise illustration of pivot and probed function values and coordinate ranks.

ation for three instances of known hard linear ordering problems[16]. Surprisingly, the *average* walkLenth under the self-avoiding walk to repeatedly find the best known solution (used as a target value in this experiment) is reported at 57,738 steps for the problem with 23! *feasible solutions*; we also observe that the best-known solution has been found in as little as 1,661 steps and that the longest walk required 231,327 steps.

The sample size of 100 is justified in order to detect non-trivial adjustments of solver heuristics. For combinatorial solvers, *uncensored*

[16] Michel X. Goemans and Leslie A. Hall. The strongest facets of the acyclic subgraph polytope are unknown. In William H. Cunningham et al, editor, *Integer Programming and Combinatorial Optimization*, volume 1084 of *LNCS*, pages 415–429. Springer, 1996





(c.1)

statsVar	solverName	instanceDim	hitRatio	cntCensored	sampleSize	median	mean	stdDev	coefVar	stdErr	meanLB95	meanUB95	minVal	maxVal
runtime	P.lopT	13	1.0000	0	100	0.038	0.06264	0.07187089	1.147364	0.007187089	0.04827228	0.07700772	0.005	0.415
cntProbe	P.lopT	13	1.0000	0	100	308	511.24	586.4265	1.147067	58.64265	394.0074	628.4726	35	3357
walkLength	P.lopT	13	1.0000	0	100	153.5	255.12	293.2132	1.149315	29.32132	196.5037	313.7363	17	1678
cntRestart	P.lopT	13	1.0000	0	100	0	0	0	NA	0	0	0	0	0
speedProbe	P.lopT	13	1.0000	0	100	8141.5	8099.48	151.9072	0.01875518	15.19072	8069.112	8129.848	7458	8280

(c.2)

statsVar	solverName	instanceDim	hitRatio	cntCensored	sampleSize	median	mean	stdDev	coefVar	stdErr	meanLB95	meanUB95	minVal	maxVal
runtime	P.lopT	19	1.0000	0	100	1.88	2.16133	1.802161	0.8338206	0.1802161	1.80106	2.5216	0.032	9.4
cntProbe	P.lopT	19	1.0000	0	100	8546	9779.6	8112.289	0.8295114	811.2289	8157.871	11401.33	147	41951
walkLength	P.lopT	19	1.0000	0	100	4272.5	4889.3	4056.145	0.8295962	405.6145	4078.436	5700.164	73	20975
cntRestart	P.lopT	19	1.0000	0	100	0	0	0	NA	0	0	0	0	0
speedProbe	P.lopT	19	1.0000	0	100	4541	4537.1	56.32024	0.01241327	5.632024	4525.841	4548.359	4118	4634

(c.3)

statsVar	solverName	instanceDim	hitRatio	cntCensored	sampleSize	median	mean	stdDev	coefVar	stdErr	meanLB95	meanUB95	minVal	maxVal
runtime	P.lopT	23	1.0000	0	100	29.2	35.8749	29.32275	0.8173612	2.932275	30.01298	41.73682	1.01	144
cntProbe	P.lopT	23	1.0000	0	100	94266	115477.6	94182.91	0.8155947	9418.291	96649.46	134305.7	3323	462655
walkLength	P.lopT	23	1.0000	0	100	47132.5	57738.29	47091.45	0.8156018	4709.145	48324.23	67152.35	1661	231327
cntRestart	P.lopT	23	1.0000	0	100	0	0	0	NA	0	0	0	0	0
speedProbe	P.lopT	23	1.0000	0	100	3229.5	3228.64	26.39809	0.008176225	2.639809	3223.363	3233.917	3160	3301

Figure 12: A simulated asymptotic performance evaluation of two combinatorial solvers, given two reference models of *cntProbe* variable for the instance set $L = \{13, 21, 27, 41, 43, 45, 51\}$: ref1 $= 500 * 1.150^L$ and ref2 $= 100 * 1.230^L$. There are $N = 100$ variates generated by each model for each L, all variates have exponential distribution. Sample means are based on 100 variates for each L, values reported for ensemble means are based on the best-fit model with respect to all variates in the instance set. Due to exponential distribution of variates, sample means are not equivalent to ensemble means which we would expect under normal or uniform variate distribution.

random variables such as *runtime* have near-exponential distribution, i.e. $s \approx \overline{m}$ where s denotes the sample standard deviation and $\overline{m}$ denotes the sample mean. In this case, a reliable rule-of-thumb estimate of the 95% confidence interval on value of the sample mean $\overline{m}$, given a sample size of 100 is $[0.8 \times \overline{m},\ 1.2 \times \overline{m}]$.

The simulated asymptotic experiments in Figures 12a and 12b demonstrate that, with *uncensored runtimes* and the sample size of 100, the *sample means* and not the *ensemble means* of each combinatorial solver produce the most reliable estimate of the asymptotic performance differences between such solvers.

References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial optimization problems and their approximability properties*. Springer Verlag, 1999.

[2] Franc Brglez. Of n-dimensional Dice, Combinatorial Optimization, and Reproducible Research: An Introduction. *Eletrotehniški Vestnik 78(4) 2011: pp. 181–192, http://ev.fe.uni-lj.si/4-2011/Brglez.pdf*, 78(4):181–192, 2011.

[3] Franc Brglez. Self-Avoiding Walks across n-Dimensional Dice and Combinatorial Optimization: An Introduction. *Informacije MIDEM, 44 (1) (2014), pp. 53-68, http://www.midem-drustvo.si/journal/*, 44(1):53–68, 2014.

[4] Borko Bošković, Franc Brglez, and Janez Brest. Low-Autocorrelation Binary Sequences: On Improved Merit Factors and Runtime Predictions to Achieve Them. *http://arxiv.org/, also under journal review*, 2015.

[5] Franc Brglez, Johnny Nguyen, and Yang Ho. Lights-out Problem and the Asymptotic Performance of Combinatorial Search under Self-Avoding Walks. *http://arxiv.org/, also under journal review*, 2015.

[6] Franc Brglez, Yang Ho, and Johnny Nguyen. Linear-Ordering Problem and the Asymptotic Performance of Combinatorial Search under Self-Avoding Walks. *http://arxiv.org/, also under journal review*, 2015.

[7] Golomb Ruler Wikipedia. http://en.wikipedia.org/wiki/Golomb_ruler, March 2015.

[8] M. D. Atkinson and Nicola Santoro. Integer Sets with Distinct Sums and Differences and Carrier Frequency Assignments for Nonlinear Repeaters. *IEEE Trans. on Comm.*, 1986.

[9] C. Cotta, . Dotu, A. J. Fernandez, and P. Van Hentenryck. A Memetic Approach to Golomb Rulers. In T. P. Runarsson et al, editor, *Parallel Problem Solving from Nature*, LNCS, pages 252–261. Springer, 2007.

[10] M.M. Polash, A. Newton, M.A. Hakim, and A. Sattar. Constraint-Based Local Search for Golomb Rulers. In Michel Laurent, editor, *Integration of AI and OR Techniques in Constraint Programming*, LNCS, pages 322–331. Springer, 2015.

[11] Michel X. Goemans and Leslie A. Hall. The strongest facets of the acyclic subgraph polytope are unknown. In William H. Cunningham et al, editor, *Integer Programming and Combinatorial Optimization*, volume 1084 of *LNCS*, pages 415–429. Springer, 1996.