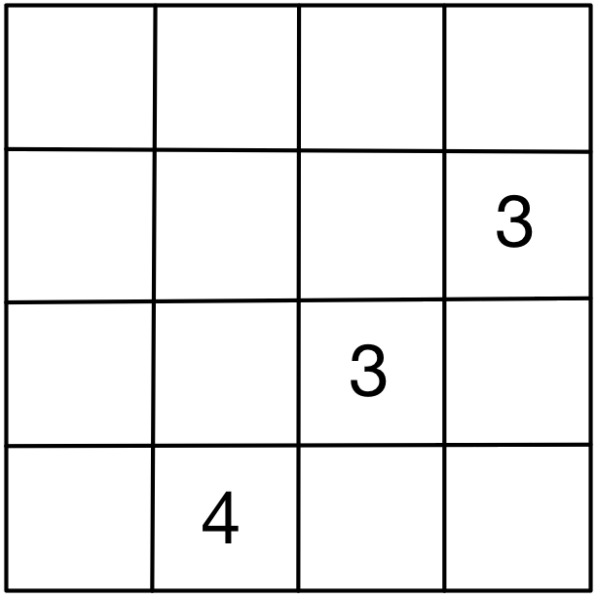
Constraint Systems

Randomization and Restarts

New Stuff from Old Stuff

Remember the PLS? It has two very intriguing properties

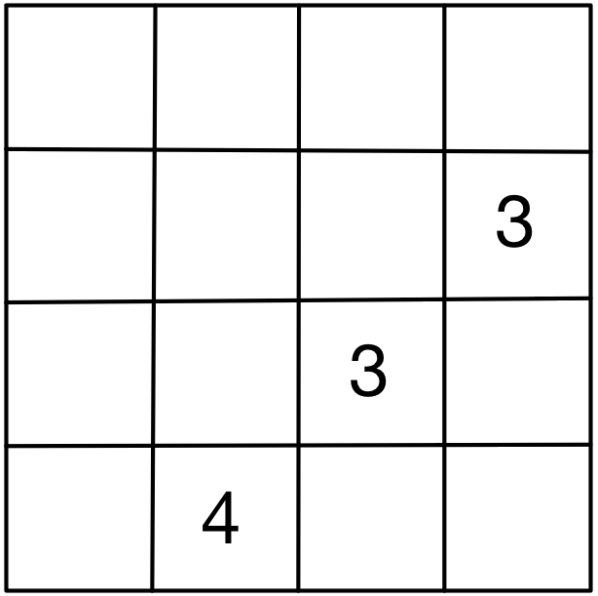


1. A phase transition
2. A heavy-tailed distribution in performance profiles

Let's start from property #1...

About Generating PLS Instances

HP: we generate PLS instances by randomly filling some cells



- If only a few cells are filled…
- …The instance will likely be feasible (and with many solutions)

About Generating PLS Instances

HP: we generate PLS instances by randomly filling some cells

3		2	
	1		3
2		3	
	4		

- If many cells are filled…
- …The instance will likely be infeasible

Phase Transitions in Combinatorial Problems

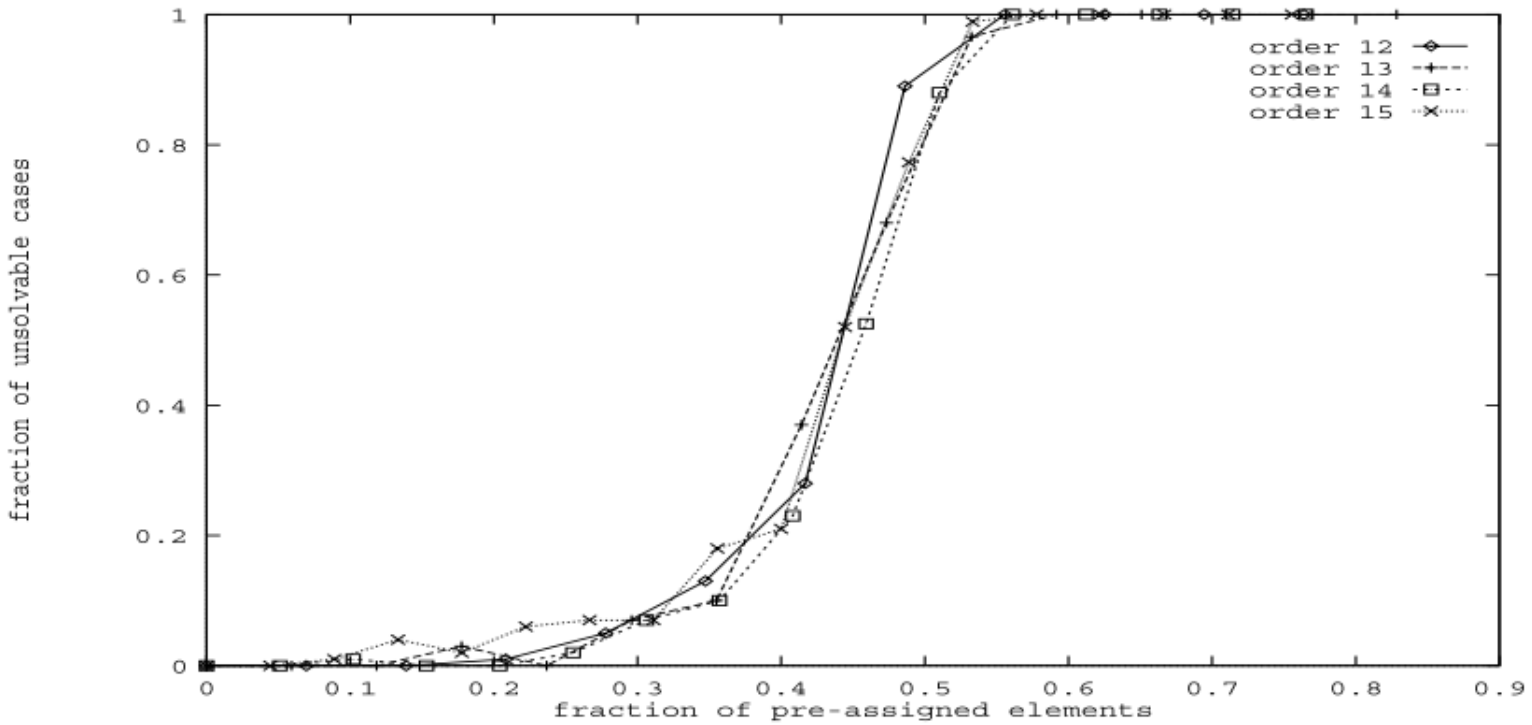
Here comes the first property:

For a certain fraction of pre-filled cells, the likelihood of having a feasible instance changes abruptly

Phase Transitions in Combinatorial Problems

The probability of having a infeasible problem has this trend:

- *Plot from: Gomes, C. P., Selman, B. & Crato, N. (1997). Heavy-tailed distributions in combinatorial search. Proc. of CP 97, 1330, 121–135.*



Phase Transitions in Combinatorial Problems

Here comes the first property:

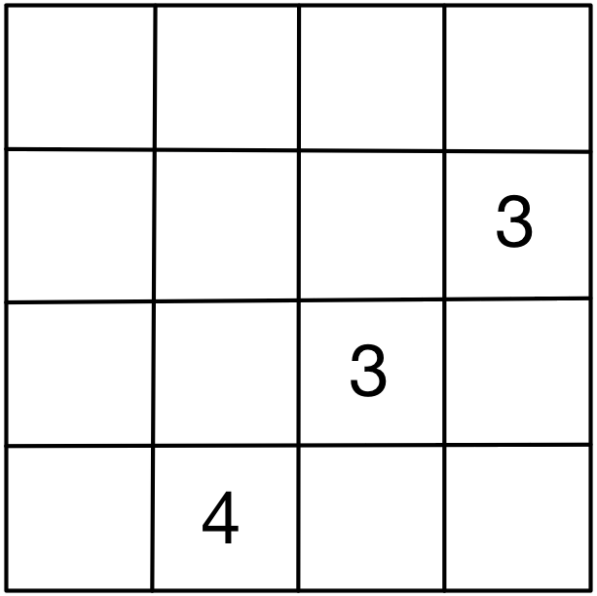
For a certain fraction of pre-filled cells, the likelihood of having a feasible instance changes abruptly

We say that the problem has a phase transition

- The term is based on an analogy with physical systems
- This is common to many combinatorial problems
  - Of course the *parameters* that control the transitions…
  - …Will be different (and likely more complex)

Phase Transitions and Difficulty

Let's see another face of the same coin:



- If only a few cells are filled
- There will likely be many solutions
- Hence, solving the problem will be easy

Phase Transitions and Difficulty

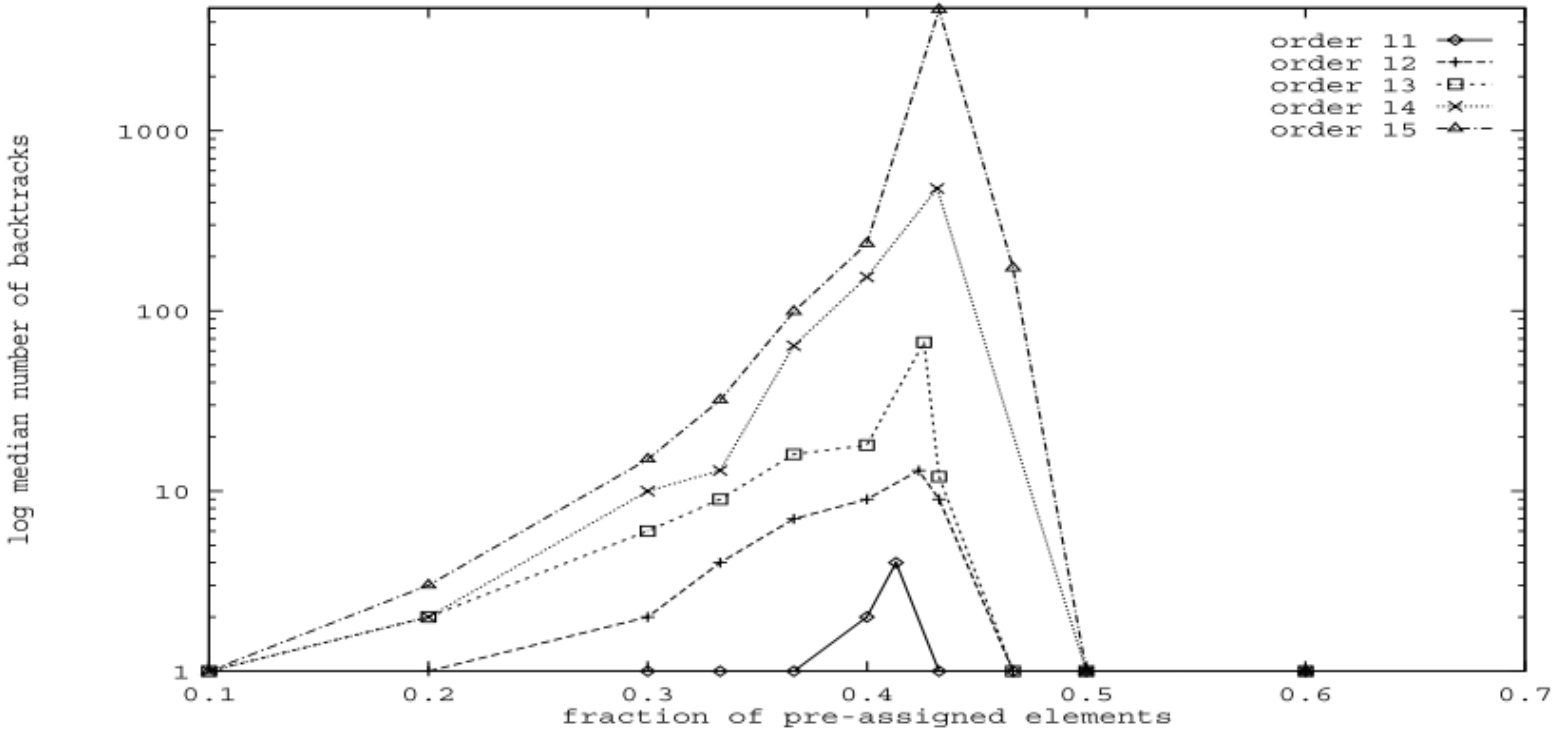
Let's see another face of the same coin:

3		2	
	1		3
2		3	
	4		

- If many cells are filled
- Constraint propagation will be very effective
- And solving the problem will be easy again

Phase Transitions and Difficulty

- The most difficult problems will lay somewhere in the middle...
- **...In fact, they lay exactly on the phase transition**



Phase Transitions and Difficulty

This is actually generalizable:

If a problem has a phase transition, the most difficult instances tend to lay on the phase transition

This holds for solution methods that are based on:

- Backtracking (which leads to threshing)
- Constraint Propagation (easy instances with many constraints)

E.g. CP, but also MILP and SAT (for those who know about them)

Phase Transitions: Wrap Up

In truth, phase transitions are properties of:

- A problem (e.g. PLS)
- An instance generation approach (e.g. randomly fill cells)
- A solution method (e.g. DFS + propagation)

Any change of those can affect the phase transition

Still, many combinatorial problems have phase transitions!

- There are some conjectures to explain this behavior…
- …Still no general explanation, however

A side note: *this is how I tuned all the instances for the lab sessions*

PLS and Search Strategies

Designing a good search strategy for the PLS is not so easy

- Using min-size-dom for the branching variable is a good idea
- Everything else is complicated

By changing the variable or value selection rule:

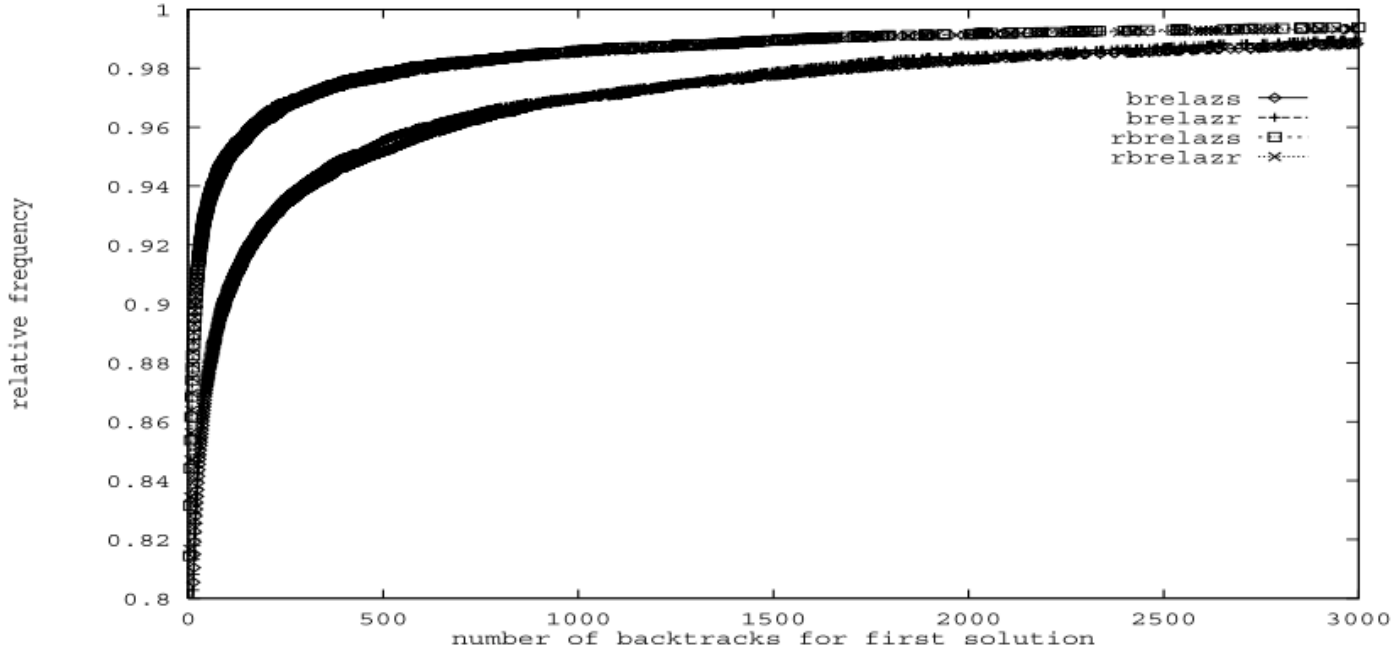
- A few hard instances become suddenly easy and vice-versa
- There are always a few difficult instances…
- …And they are not always the same ones!

You may have observed this behavior in the lab

It makes tuning the selection heuristics kind of frustrating

PLS and Search Strategies

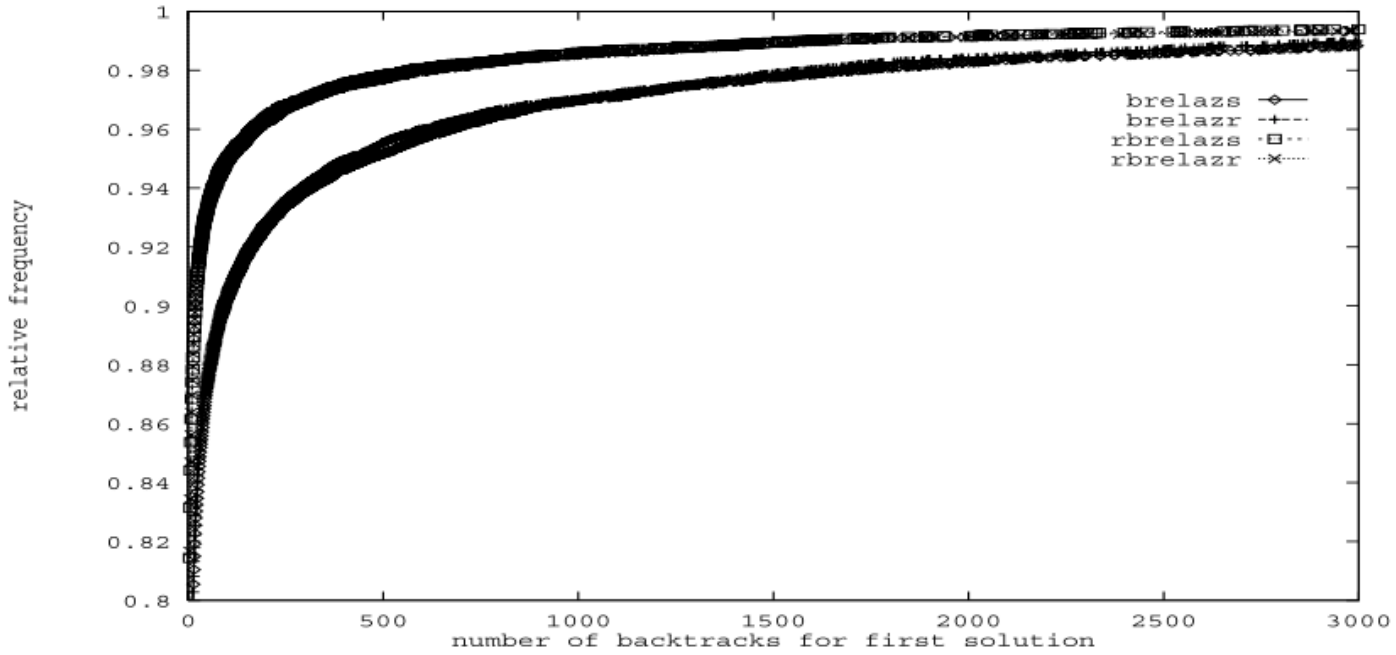
Here's another plot from the Gomes-Selman paper:



- Each curve = a different tie braking rule for min-size-dom
- $y =$ number of problems solved with $\leq x$ fails

PLS and Search Strategies

Here's another plot from the Gomes-Selman paper:



- Most instances are solved with a few backtracks
- A few instances take much longer

Randomized Search Strategies

In summary, if we slightly alter a good var/val selection heuristic

- The general performance stays good...
- ...But suddenly hard instances become easy...
- ...And some easy instances become hard

This behavior is common to many combinatorial problems

Intuitively, the reason is that:

- If we make a mistake early during search, we get stuck in thrashing
- Different heuristics lead to "bad" mistakes on different instances

A big issue: such mistakes are seemingly random

An (apparently) crazy idea: can we make this a asset?

Randomized Search Strategies

Let us assume to randomize the var/val selection heuristics:

- Pick a variable/value at random
- Randomly break ties
- Pick randomly among the 20% best
- …

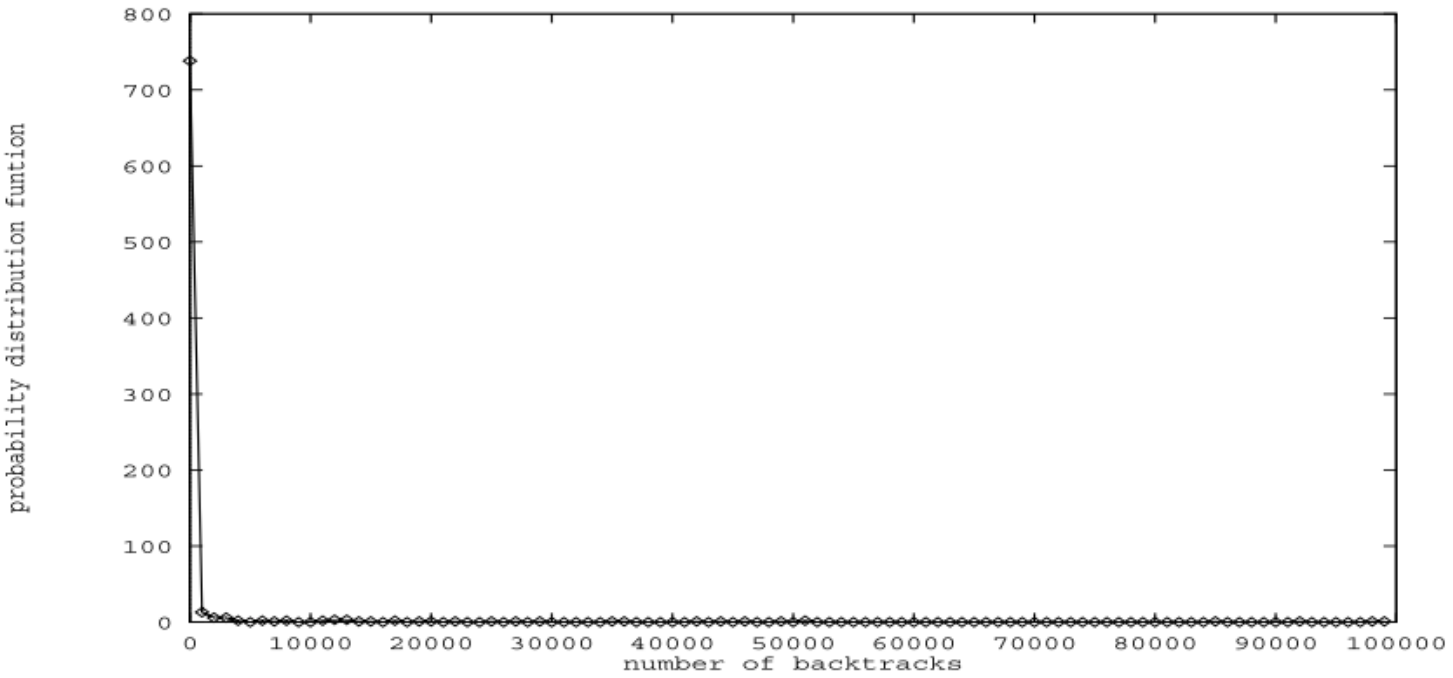
Some notes:

- We are still complete (we can explore the whole search tree)
- But the solution method becomes stochastic!
- Multiple runs on the same instance yield different results

Can we say something about the "average" performance?

Randomized Search Strategies

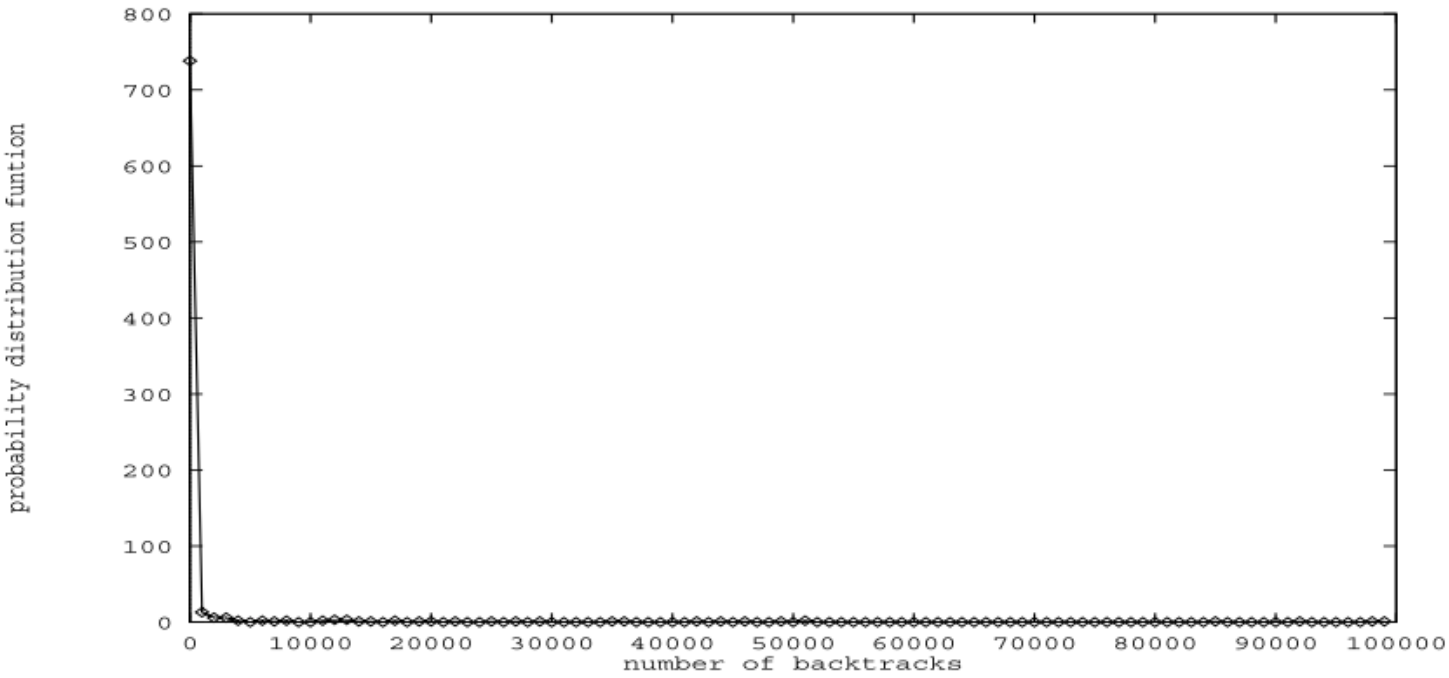
We can do more: i.e. plot an approximate Probability Density Function:



- $y =$ probability to solve an instance with x backtracks
- The plot is for a single instance
- It gives an idea of how lucky/unlucky we can be

Randomized Search Strategies

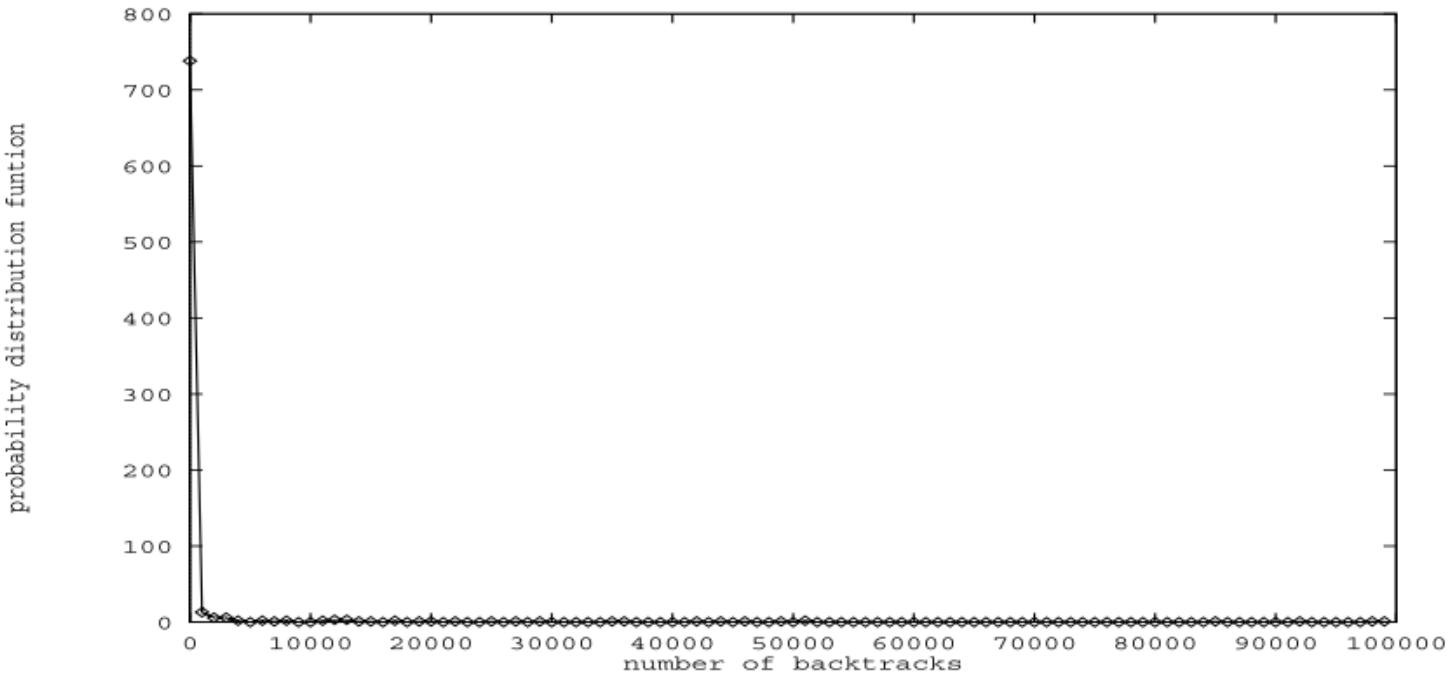
We can do more: i.e. plot an approximate Probability Density Function:



- There is high chance to solve the instance with just a few backtracks
- There is a small, but non-negligible chance to branch much more

Randomized Search Strategies

We can do more: i.e. plot an approximate Probability Density Function:



In other words, it's the same situation as before

- Instead of random instances, we have a randomized strategy...
- ...But we have the same statistical properties

Heavy-Tailed Behavior

We say that the performance has a heavy-tailed distribution

- Formally: the tail of the distribution has a sub-exponential decrease
- Intuitively: you will be unlucky in at least a few cases

In practice:

For a deterministic approach and random instances:

- There are always a few instances with poor performance

For a stochastic approach and a single instance:

- There are always a few bad runs

So far, it doesn't sound like good news...

(Random) Restarts

However, when we have a heavy-tailed distribution:

We can both improve and stabilize the performance by using restarts

- We start to search, with a resource limit (e.g. fails or time)
- When the limit is reached, we restart from scratch

The guiding principle is: "better luck next time!"

- Same as the state lottery :-)
- Except that here it works very well
- Because there is a high chance to be lucky

Restarts and Complete Search

By restarting we do not (necessarily) loose completeness

...We just need to increase the resource limit over time:

1 fail, 100 fails, 200 fails, …

The law used to update the limit is called restart strategy

We may waste some time...

- ...Because we may re-explore the same search space region
- But not necessarily: there are approaches that, before restarting...
- ...Try to learn a new constraint that encodes the reason for the failure
- This is called nogood learning (we will not see the details)

In general restarts are often very effective!

Restart Strategies

There are two widely adopted restart strategies

Luby strategy:

$$1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$$

- A 2 every two 1s
- A 4 every two 2s
- An 8 every two 4s
- And so on and so forth

This strategy has strong theoretical convergence properties

- It is guaranteed to be within a logaritmic factor from optimal

Restart Strategies

There are two widely adopted restart strategies

Walsh strategy (geometric progression):

$$1, r, r^2, r^3, \ldots$$

- with $r > 1$ (typically $r \in (1, 2]$)

This strategy may work better than Luby's in practice

In both cases, it is common to add a scaling factor

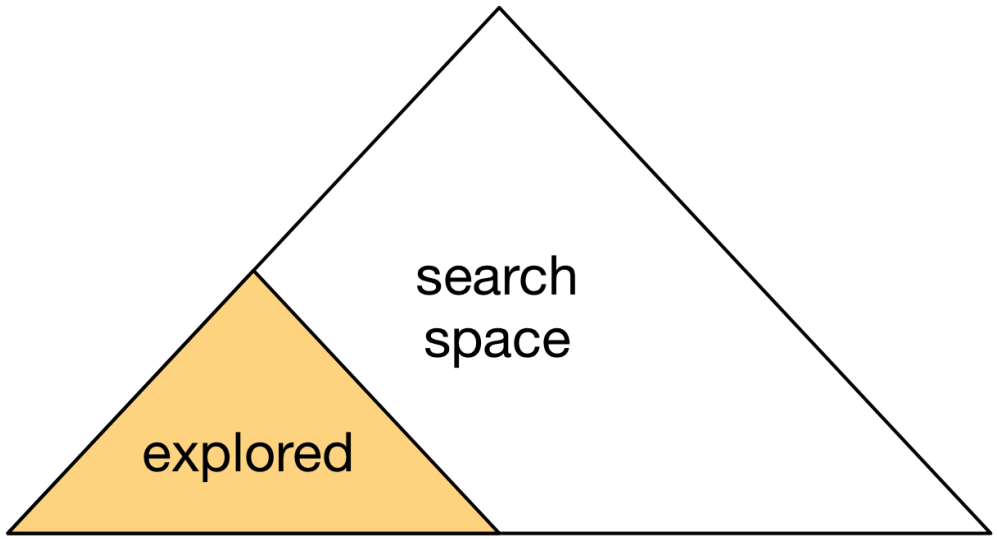
- Scaled Luby's: $s, s, 2s, s, s, 2s, 4s, \ldots$
- Scaled Walsh: $s, s\,r, s\,r^2, s\,r^3, \ldots$

Restarts and Large Scale Problems

Restarts help with large scale problems:

- Large scale problems are difficult to explore completely
- Usually a global time/fail limits is enforced

Without restarts, we obtain this behavior:



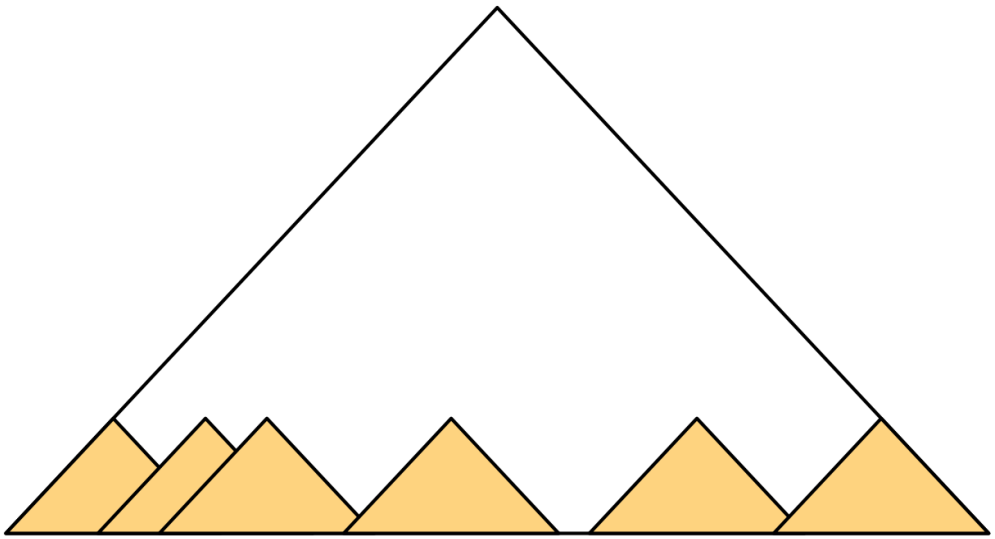
- Yellow area = region that we manage to explore within a time limit

Restarts and Large Scale Problems

Restarts help with large scale problems:

- Large scale problems are difficult to explore completely
- Usually a global time/fail limits is enforced

With restarts, instead we have this:



Restarts and Large Scale Problems

Restarts help with large scale problems:

- Large scale problems are difficult to explore completely
- Usually a global time/fail limits is enforced

Using restarts, we explore the search tree more uniformly

- This is definitely a good idea!
- Unless we have an extremely good search strategy…

It works well for optimization problems, too!

- Every time we find an improving solution we get a new bound
- The bounds may guide the search heuristics in later attempts

Restarts may increase the time for the proof of optimality

Constraint Systems

Large Neighborhood Search

Local Search

A classical approach for large-scale optimization problems:

Local Search (Hill Climbing)

σ = initial solution
while true:
    $\sigma' = explore(N(\sigma))$
    if no improving solution σ' is found: break
    $\sigma = \sigma'$

- We start from a feasible solution σ
- We search for a better solution σ' in a neighborhood $N(\sigma)$
- If we find one, σ' becomes the new σ and we repeat

Main underlying idea: high quality solutions are likely clustered

Local Search

Local Search works very well in many cases

- LS is scalable
  - $N(\sigma)$ is often defined via simple moves (e.g. swaps)
  - Hence, $N(\sigma)$ is typically small
- It is an anytime algorithm (always returns a feasible solution)

Main drawback: LS can be trapped in a local optimum

This can be addressed via several techniques, e.g.:

- Accept worsening moves (e.g. simulated annealing, Tabu Search)
- Keep multiple solutions (e.g. Genetic Alg., Particle Swarm Opt.)
- Randomization (e.g. Ant Colony Opt., Simulated Annealing)

Large Neighborhood Search

A simpler alternative: use a larger neighborhood

Main issue: the neighborhood size grows exponentially

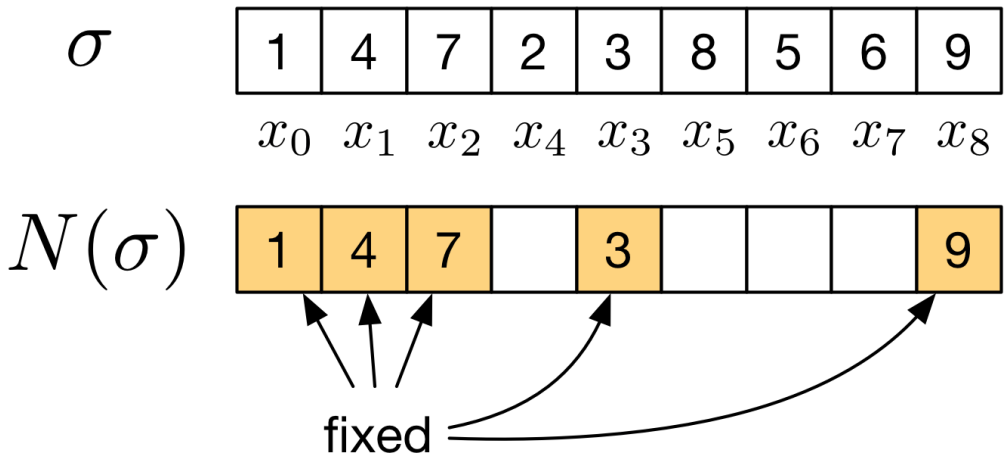
- E.g. Swap pairs: $\sim n^2$, swap triples: $\sim n^3$

A solution: use combinatorial optimization to explore $N(\sigma)$

- We can use CP, or Mixed Integer Linear Programming, or SAT!
- We will consider the CP case

Large Neighborhood Search

How do we define the neighborhood in this case?



- Fix part of the variables to the values they have in σ
- Relax (i.e. do not pre-assign) the remaining variables

The set of fixed values is sometimes called a fragment

Large Neighborhood Search

This approach is know as Large Neighborhood Search

Here's the pseudo code for a basic version

$P = \langle X, D, C \rangle$
while stop condition not met:
    $P' = P$
    for $i \in fragment(P')$:
        add the constraint $x_i = \sigma(x_i)$ to P'
    $\sigma' = solve(P')$
    if an improving solution σ' has been found: $\sigma = \sigma'$

Iteratively:

- We define a subproblem by fixing variables
- We solve the subproblem
- Possibly, we move to a new solution

Advantages of Using LNS

Using LNS has several advantages

1) LNS enables the use of large neighborhoods:

- Thanks to propagation and advanced search strategies
- In principle, you can do something similar in a custom LS approach
  - But you end up coding a small CP solver!

2) LNS is easier to develop than Local Search

- It's easy to define a neighborhood: just fix some vars
- No need to ensure that complicated constraints are satisfied
  - CP takes care of this
- Of course, you need an underlying CP solver

Advantages of Using LNS

Using LNS has several advantages

3) It's more scalable than running CP on the whole problem

- The sub-problems are typically much smaller!
- And we can control the sub-problem size
  - Of course if they are too small we may get stuck in local optima

4) Each sub-problem is explored more effectively

- Propagation works best when the domains are small
- The fixed variables in the sub-problem reduce the domain sizes

LNS Parameters

LNS is a heuristic approach

- No proof of optimality (as a rule)
- Many tuning parameters and design decisions, like most heuristics

Here are the most important design decisions:

1) Complete vs incomplete neighborhood exploration

- In LS, each neighborhood is always completely explored
- In LNS, it is often useful to allow partial exploration
- This is done by enforcing a resource limit (time or fails)
- Typically: tuned to have a > 50% chance of complete exploration

LNS Parameters

LNS is a heuristic approach

- No proof of optimality (as a rule)
- Many tuning parameters and design decisions, like most heuristics

Here are the most important design decisions:

2) How many improving solutions?

- Typically, either stop at the first improving solution
- Or keep on exploring until the resource limit is reached

3) Which and how many variables to relax?

- This is by far the most important design choice
- And requires a deeper discussion...

Fragment Selection in LNS

About choosing the variables to relax:

1) Consider random selection as a baseline

- Can work surprisingly well!
- Ensures diversification: explore different search space areas

2) Problem-specific approaches

- E.g. all items assigned to certain bins
- E.g. relax one day in a schedule

3) Automatic/adaptive Techniques

- Propagation-based: we will see an example
- Cost-based, learning-based: see paper on the course web site

An Example of Automatic Fragment Selection

We will now see an example of automatic fragment selection

A state-of-the-art approach proposed in:

> *Perron, L., Shaw, P., and Furnon, V. (2004). Propagation guided large neighborhood search. Proc. of CP04 (pp. 468–481). Springer*

The authors were from the IBM-ILOG CPO commercial solver

- PRO: the approach works well on practical problems
- CON: some details are not well described!

Main idea: using propagation to guide fragment selection

- We'll discuss the paper contributions one by one

Propagation-based Neighborhood Size

The first contribution is about tuning the neighborhood size

Typical approach: choose a number of variables to relax/fix

- However, because of propagation that occurs after fixing…
- …The size of the search space in the sub-problem may vary wildly

It is difficult to ensure that a sub-problem is sufficiently well explored

Can we ensure a more uniform size?

Propagation-based Neighborhood Size

Here's the approach proposed in the paper:

Step #1: perform propagation while fixing

for i in selected fragment:
    add constraint $x_i = \sigma(x_i)$ to P'
    propagate until the fix-point is reached (new contribution)

Step #2: fix as long as the search space size is above a threshold

- We can measure the size of the Cartesian product of the domains

$$\prod_{x_i \in X} |D(X_i)| > \theta$$

- Works if the product gives a good estimate of the search space

Propagation-based Variable Choice

Second contribution: using propagation for selecting vars

Step #1: Keep a list L of non-fixed variables

- Initially, L is empty
- Whenever we fix a variable, we propagate..
- ...And we measure the domain reduction for all non-fixed variables

$$score_i = 1 - \frac{|D(x_i)|_{after}}{|D(x_i)|_{before}}$$

- We insert in L all variables with $score_i > 0$
- If more than n vars are in L, we keep those with the highest score

Propagation-based Variable Choice

Second contribution: using propagation for selecting vars

Step #2: Use L for selecting the variables to fix

- If L is empty, we choose a variable at random
- Otherwise, we choose the variable from L

Some unclear points:

- How is the variable chosen from L?
  - Conjecture: the one with the largest score
- What if we have propagation on a variable already in L?
  - Conjecture: keep the highest score

These steps are unfortunately unclear from the paper

Propagation-Guided LNS

This approach is called Propagation-Guided LNS (PGLNS)

Underlying rationale:

- HP: variables form clusters, connected by tighter constraints
- It may be a good idea to fix whole clusters
- The relaxed variables will be part of the remaining clusters

Propagation-Guided LNS

This approach is called Propagation-Guided LNS (PGLNS)

Underlying rationale:

- HP: variables form clusters, connected by tighter constraints
- It may be a good idea to fix whole clusters
- The relaxed variables will be part of the remaining clusters

That makes sense, but it's a kind of a bet:

- If we find two variables that are strongly correlated...
- ...Is it really best to choose them for fixing...
- ...when we could instead relax them?

This second approach is more likely to relax whole clusters!

Reverse PGLNS

The authors call this approach reverse PGLNS

The main idea is choosing the variables to be relaxed

- The list L contains again the candidate variables

```
while fragment size < θ:
    if |L| = 0:
        choose x_i at random
    else:
        choose x_i in L
    add x_i to the set of relaxed variables
    update the list L
```

- Once the sub-problem search space is large enough…
- …We fix all the remaining variables

Reverse PGLNS

There is one main difficulty:

- Since we are relaxing variables rather than fixing them…
- …We cannot measure the domain size, and hence $score_i$!

The proposed solution:

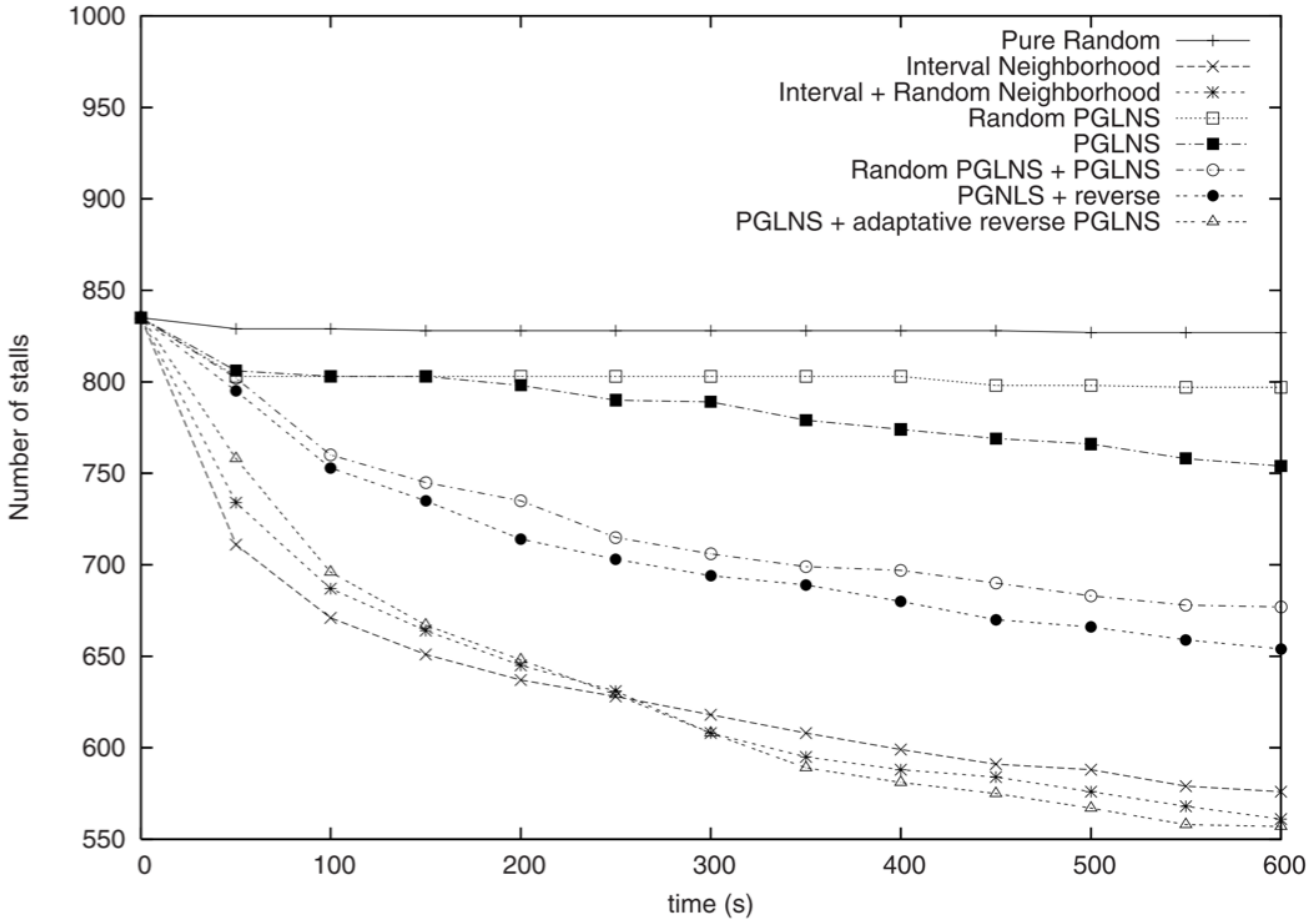
- Interleave PGLNS and reverse-PGLNS
- Use the average $score_i$ from past PGLNS iterations
- Use a similarly adjusted coefficient to estimate the domain sizes

This is (more or less) the approach proposed in the paper

- It makes use of multiple types of neighborhood
- Using multiple neighborhood types is common in practice

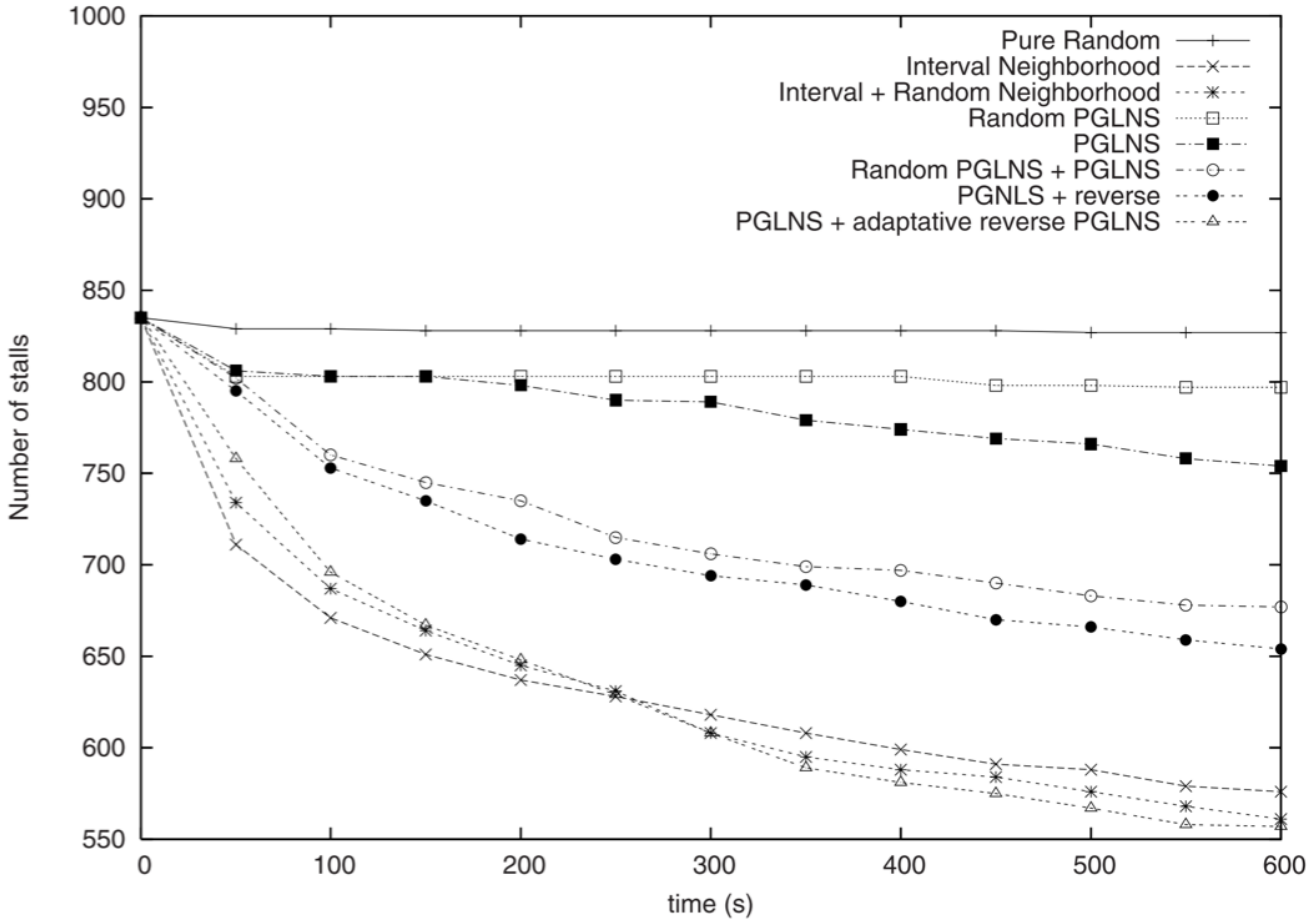
Some Results

Problem: Car Sequencing (manage cars on an assembly line)



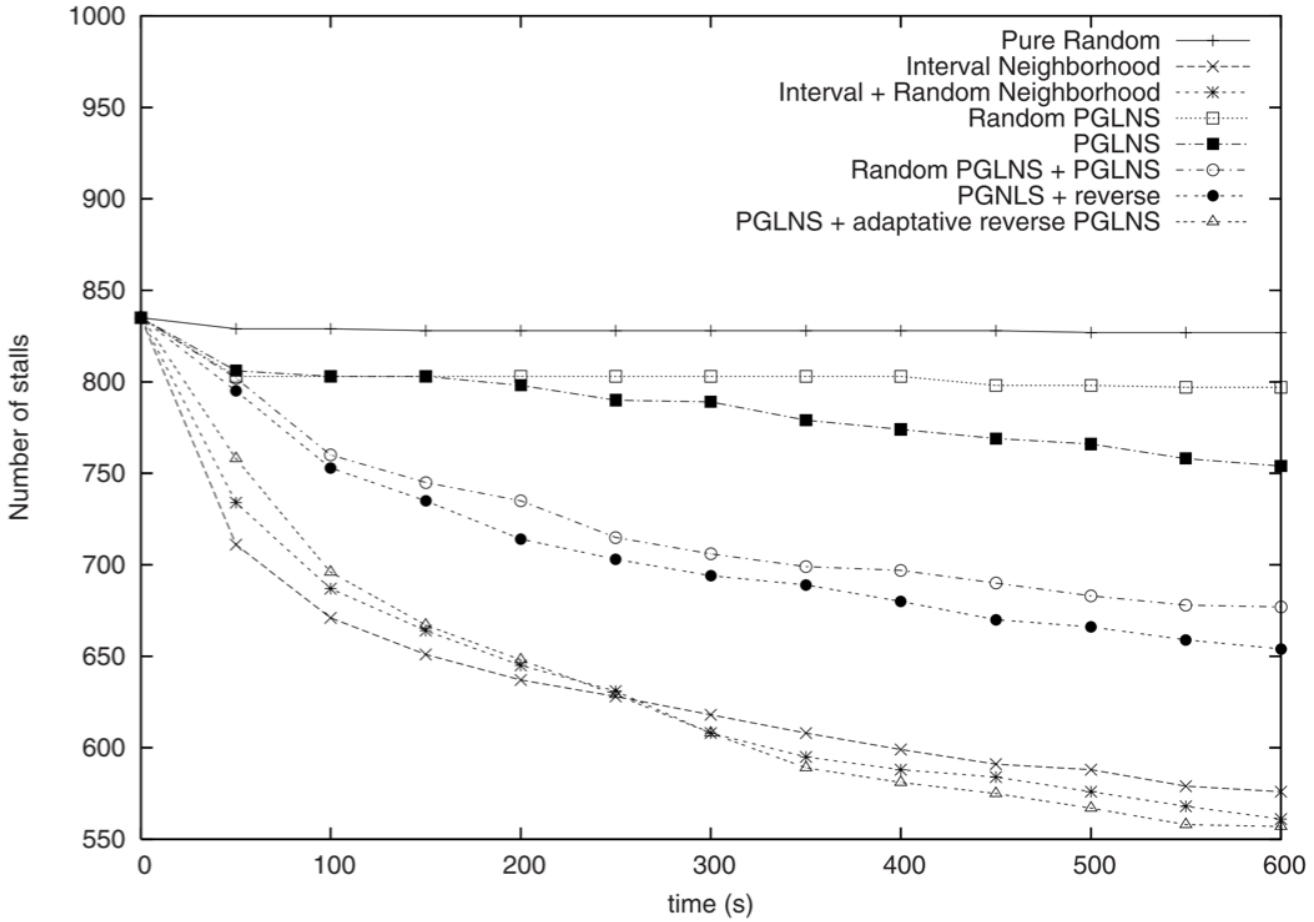
Some Results

Objective: the lower the better



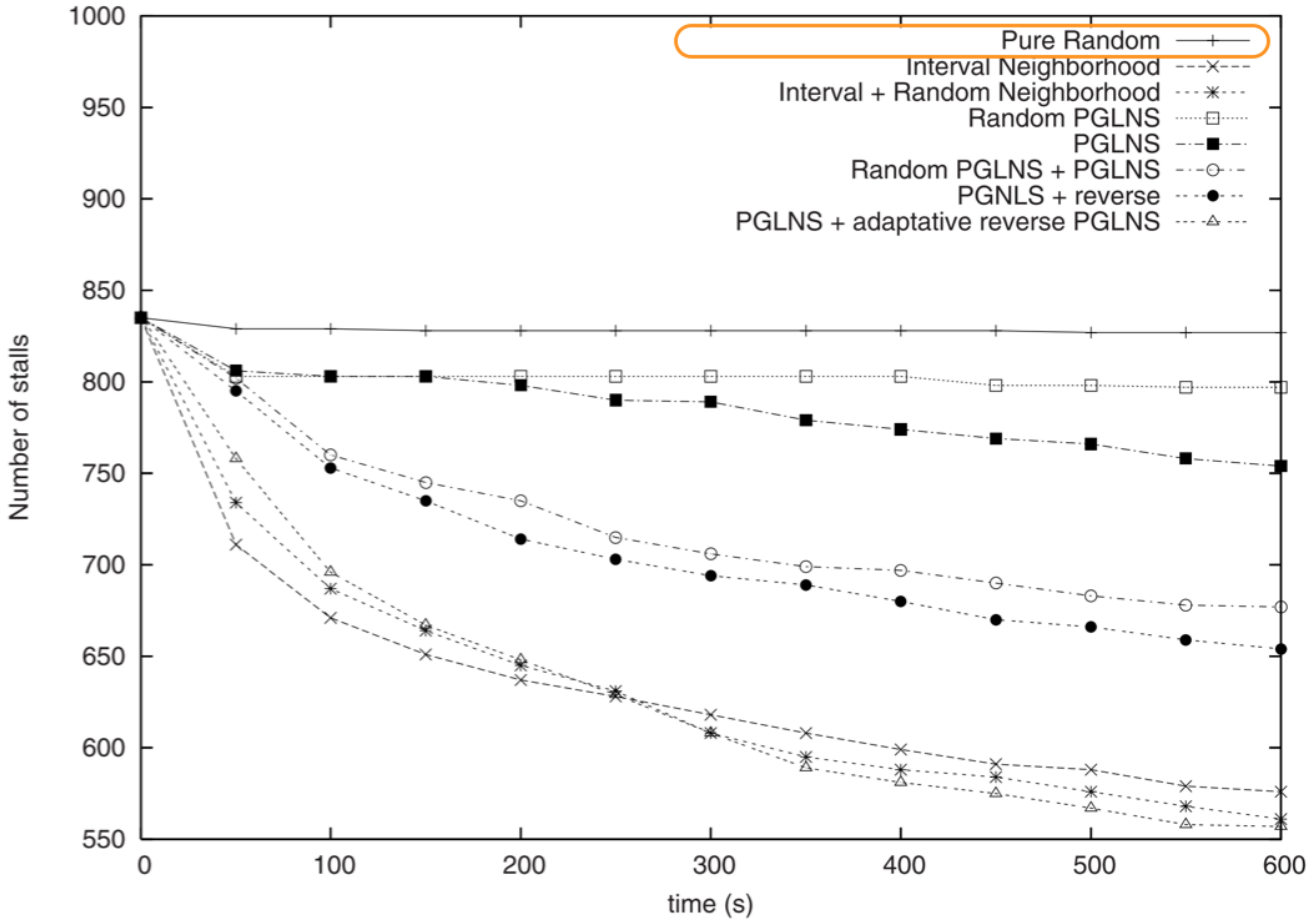
Some Results

Size: 500 time slots



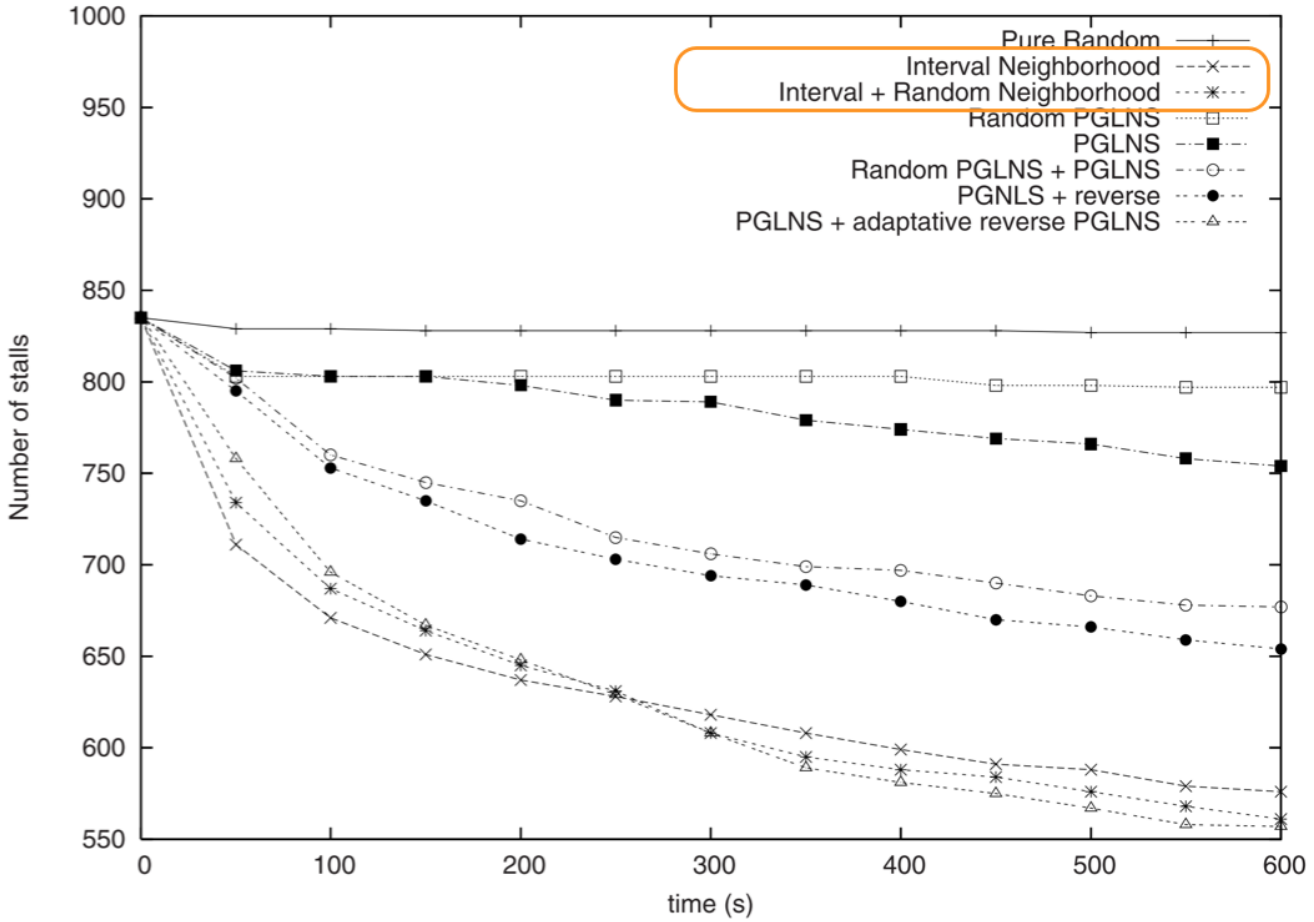
Some Results

Randomly select variables (works poorly here)



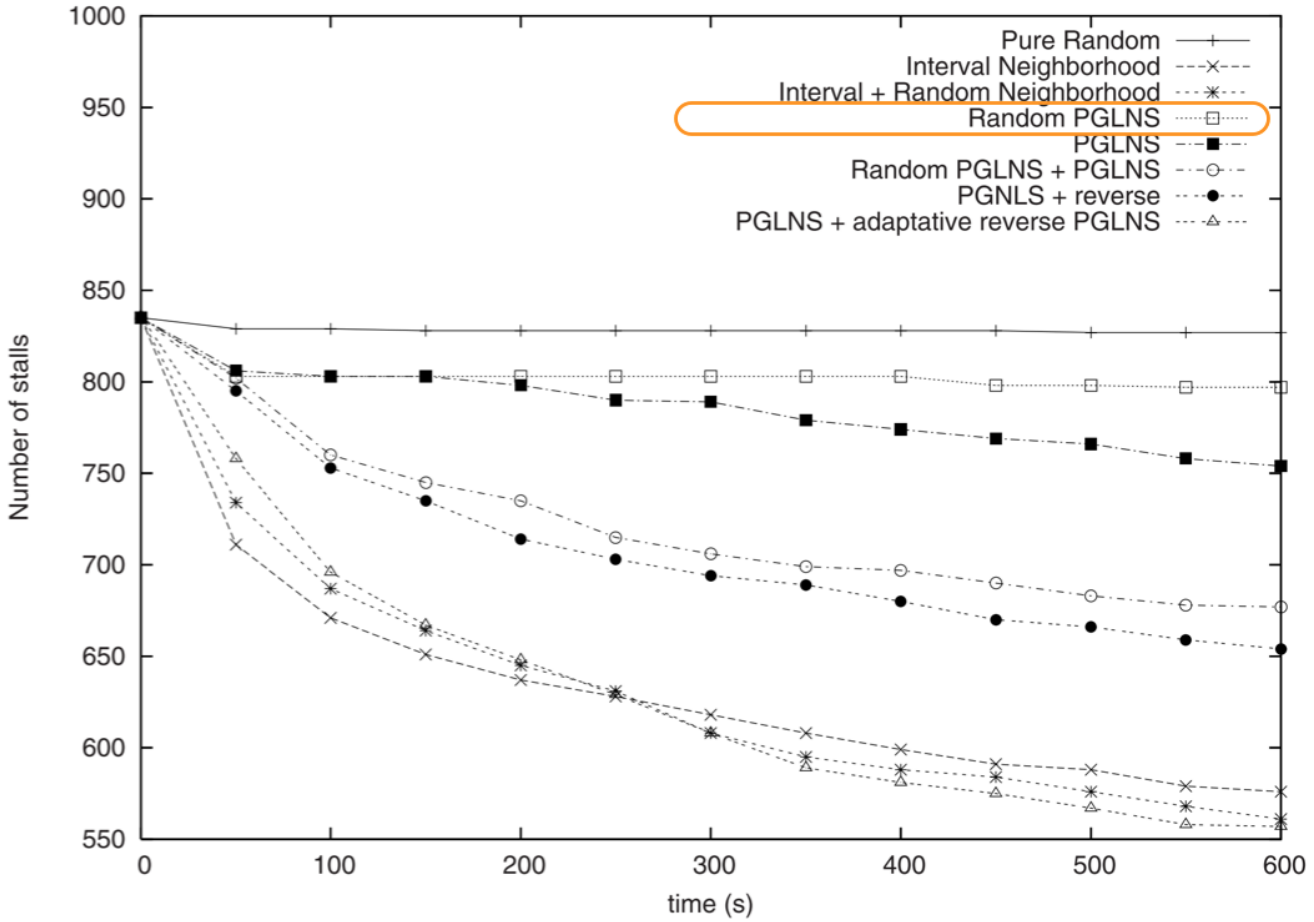
Some Results

Application specific neighborhoods (very effective!)



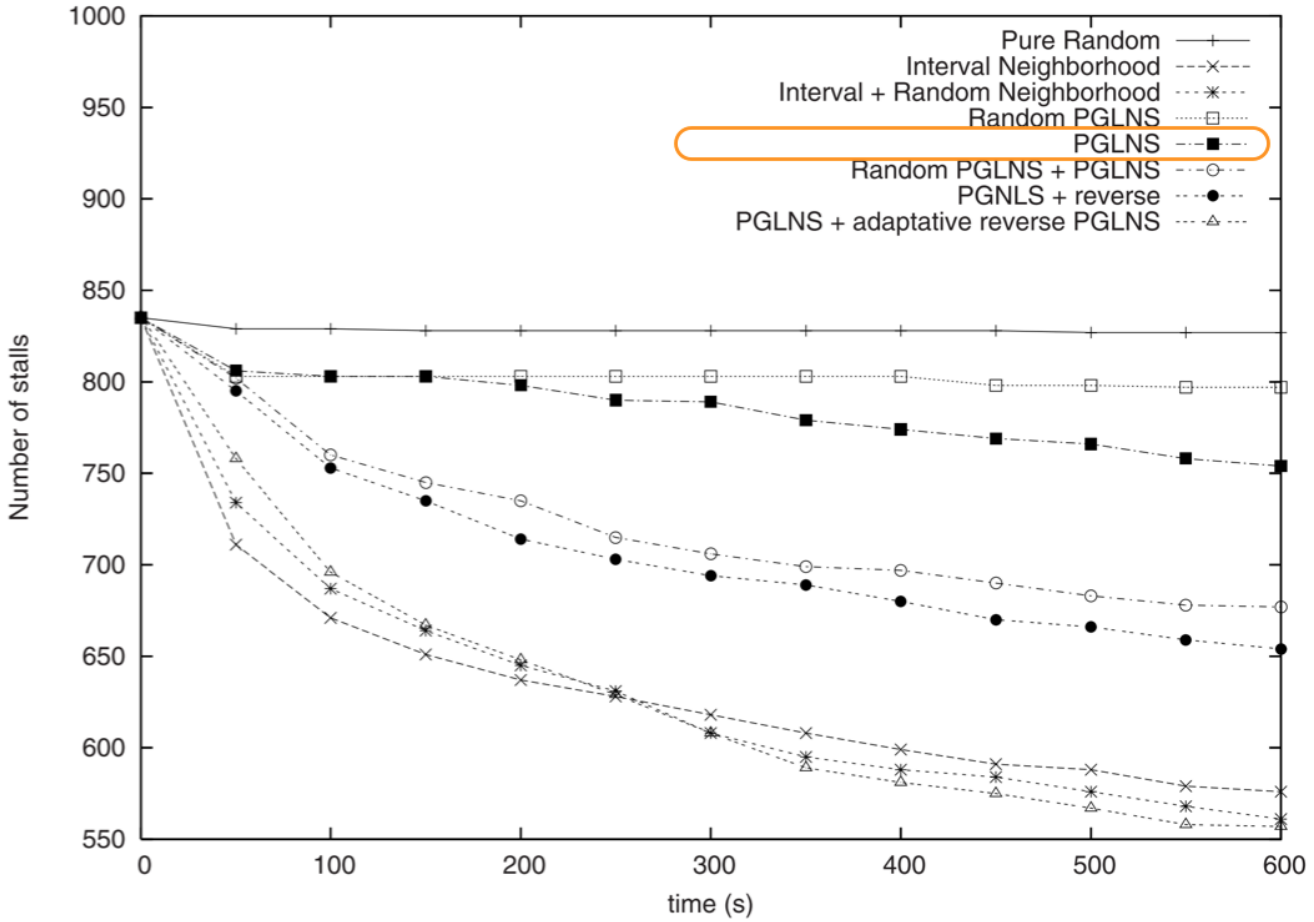
Some Results

Random selection + propagation based size control



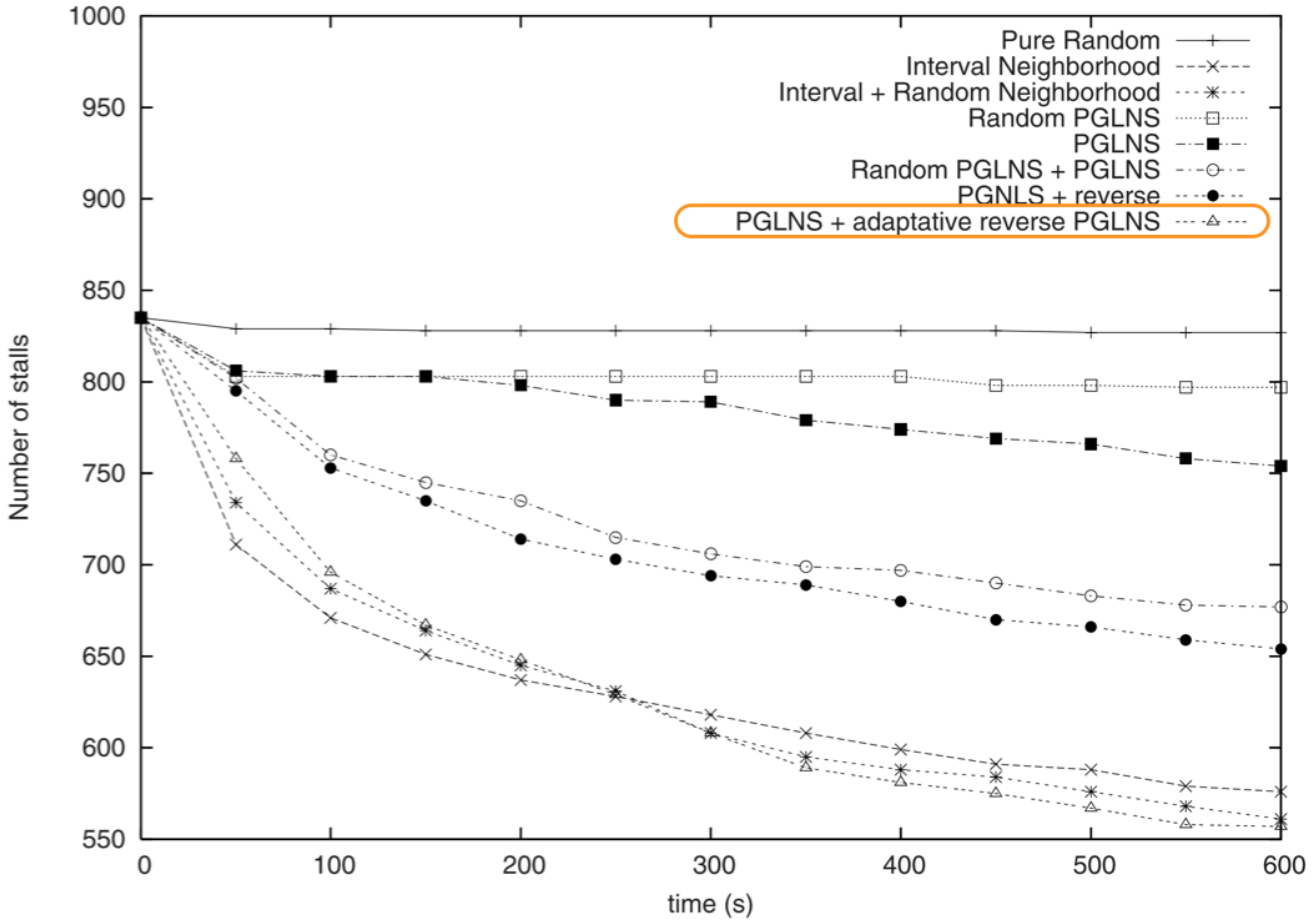
Some Results

PGLNS alone



Some Results

PGLNS + reverse PGLNS (with estimated domain sizes)



Constraint Systems

Advanced Search Heuristics

About General Search Heuristics

Traditionally, the best CP search heuristics are ad hoc made

Some PROs and CONs:

- PRO: in general, the best results require always some customization
- PRO: customizing a search strategy in CP is easy
- CON: the customization requires some CP expertise
- CON: finding a good strategy may be difficult
- CON: poor performance without customization

Other approaches (MILP, SAT...) have powerful general heuristics:

- They may leave room for improvement on some problems
- But they work very well out of the box!

About General Search Heuristics

Luckily, the picture is changing

Several general search heuristics have been proposed

- Impact Based Search (2003)
- Domain over Weighted Degree (2004)
- Counting-based heuristics (2007)
- Last Conflict(s) (2009)
- Activity based search (2012)
- Conflict Ordering Search (2015)
- Failure Directed Search (2015)
- ...And I have probably missed something

Many of these heuristics work well on a wide range of problems

General Search Heuristics: Main Ideas

Those search heuristics are based on some key ideas:

1) Learning from past propagation

- When we assign a value we reduce the search space size
- We can use this information for branching
- Typically, we apply the first-fail principle:
  - E.g. choose the variable that led to strongest propagation

2) Learning from past fails

- Can be considered a sub-case of the former
- We store information specifically from fails
- First fail principle: choose variables that caused fails

General Search Heuristics: Main Ideas

Those search heuristics are based on some key ideas:

3) Exploit information about the constraints

- Propagation or fail information is assigned to vars/vals
- This is done by looking at the constraints they are involved in

4) Extract information from constraints

- Use specialized algorithms to obtain additional information
- E.g. algorithms to estimate the number of solutions of a cst.
- Scores are assigned to vars by exploiting the constraint network

A Recent General Search Heuristic

As an example we will see Failure Directed Search

- Discussed in: *Vil, P., Laborie, P., & Shaw, P. (2015). Failure-Directed Search for Constraint-Based Scheduling, 9075, 437–453*
- Used in IBM-ILOG CPO since version 12.6

Main idea: learn from past propagation and fails

FDS is designed for infeasible problems

- E.g. the optimality proof after using LNS

Two main contributions:

- A generic search method to deal with binary decisions
- The actual search heuristic

The Search Method

The search method used by FDS is based on binary decisions

Each decision has a generic + and − branch

- E.g. $+ : x_i = v$ and $- : x_i \neq v$
- E.g. $+ : x_i \leq v$ and $- : x_i > v$

An important assumption:

We have stored a pool of possible decisions

- E.g. $+/- : x_i = v \,/\, x_i \neq v$ for all x_i and v
  - In other words: all possible assignment decisions!
  - Even using a subset is ok, however (see later)

The Search Method

Here's the pseudo-code for the search method:

```
P = initial pool of decisions
while |P| > 0:
    choose a decision
    if the + or − constraint is satisfied:
        break (i.e. the decision is "already taken")
    generate + and − search nodes and propagate
    if both nodes fail:
        backtrack
    move to one of the child nodes
```

When the pool of decisions is over:

- If all search nodes have failed, we stop
- Otherwise, we generate a new pool and repeat

The Search Method

Here's the pseudo-code for the search method:

```
P = initial pool of decisions
while |P| > 0:
    choose a decision
    if the + or − constraint is satisfied:
        break (i.e. the decision is "already taken")
    generate + and − search nodes and propagate
    if both nodes fail:
        backtrack
    move to one of the child nodes
```

Two main design parameters:

- How to pick a decision?
- How to pick one of the two branches?

The Search Method

This is very similar to the usual DFS search in CP

There are two main differences:

1) We have an explicit pool of possible decisions

2) We apply propagation immediately to both branches

Why are they introduced?

- We need (1) to keep an updated score for each decision
- We need (2) because the score is propagation-based

Computing the FDS scores

The scores are based on measure of the propagation impact

Here, we will call this measure reduction :

$$R = \frac{\prod_{x_i \in X} |D(x_i)|_{after}}{\prod_{x_i \in X} |D(x_i)|_{before}}$$

- R = ratio between search space size before and after propagation
- Lower R values correspond to stronger propagation

Side node: R is related to another measure called impact:

$$I = 1 - R$$

- Impacts were historically introduced before R (in 2003)

We will use $\mathbf{R}$ to assign a score $\mathbf{s^+}$ and $\mathbf{s^-}$ to each branch

- The score of a decision is $s^+ + s^-$

Computing the FDS scores

The FDS scores are computed as follows:

- Initially, all scores are 1
- When a branch is first processed, the score $s^{+/-}$ is equal to:

$$\begin{cases} 0 & \text{if the branch fails} \\ 1 + R & \text{otherwise} \end{cases}$$

We call this expression $localscore$

- Smaller $localscore$ values correspond to stronger propagation
- Thanks to the "+1" term...
- ...An immediate fail receives a much smaller $localscore$

Computing the FDS scores

The FDS scores are computed as follows:

- When a branch is processed again, the score is:

$$s = \alpha\, s + (1 - \alpha)\frac{localscore}{depthscore}$$

The α parameter ensures that part of the old score is retained:

- α ranges in $[0, 1]$, with typical values in $[0.9, 0.99]$
- Despite the high α values, the scores are replaced quickly!

Computing the FDS scores

The FDS scores are computed as follows:

- When a branch is processed again, the score is:

$$s = \alpha\, s + (1 - \alpha)\frac{localscore}{depthscore}$$

$depthscore$ is the average decision score at this depth

- Depth = number of taken branches
- Initially, $depthscore = 1$

Why is $depthscore$ needed?

- Propagation is more effective at high depth
- Using $depthscore$ for normalization leads to fairer results

FDS: Branch and Decision Selection

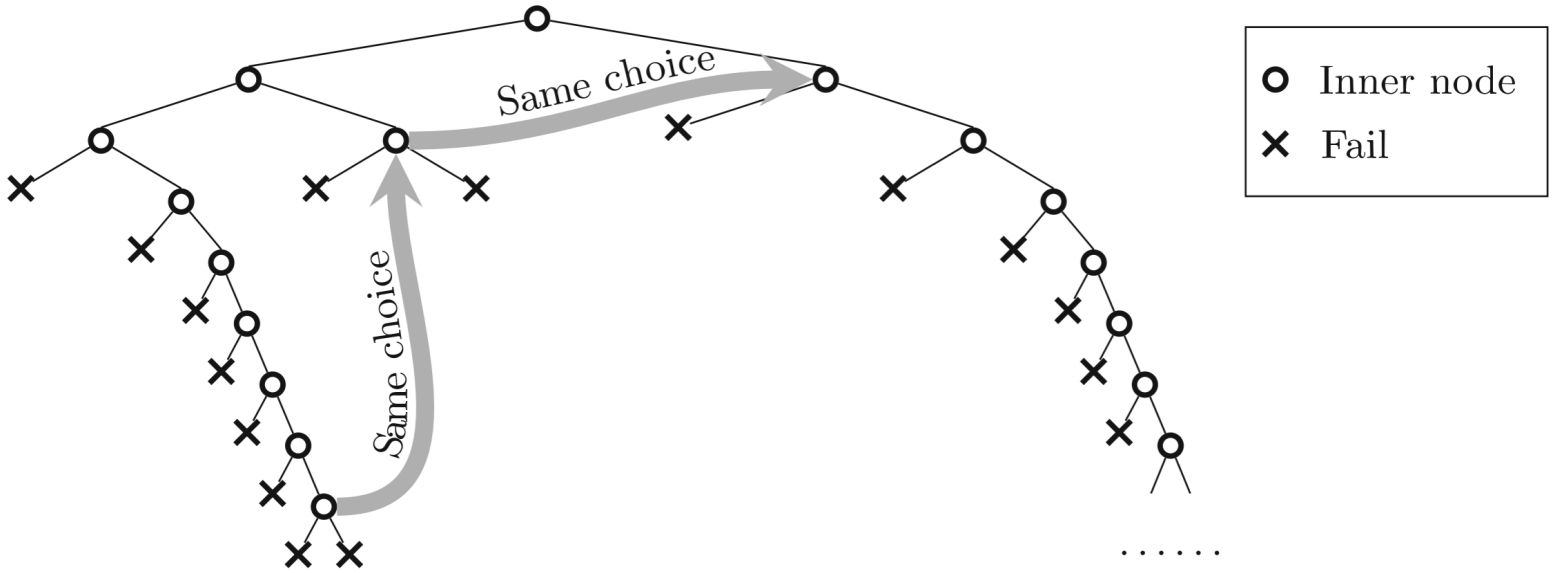
How FDS selects decision and branches:

- FDS always selects the decision with the lowest score
- FDS always selects the branch with the lowest score
- Lowest score = strongest propagation

The reason: FDS is designed for infeasible problems

- We want to maximize the probability to fail…
- …Because we want to keep the search tree as small as possible

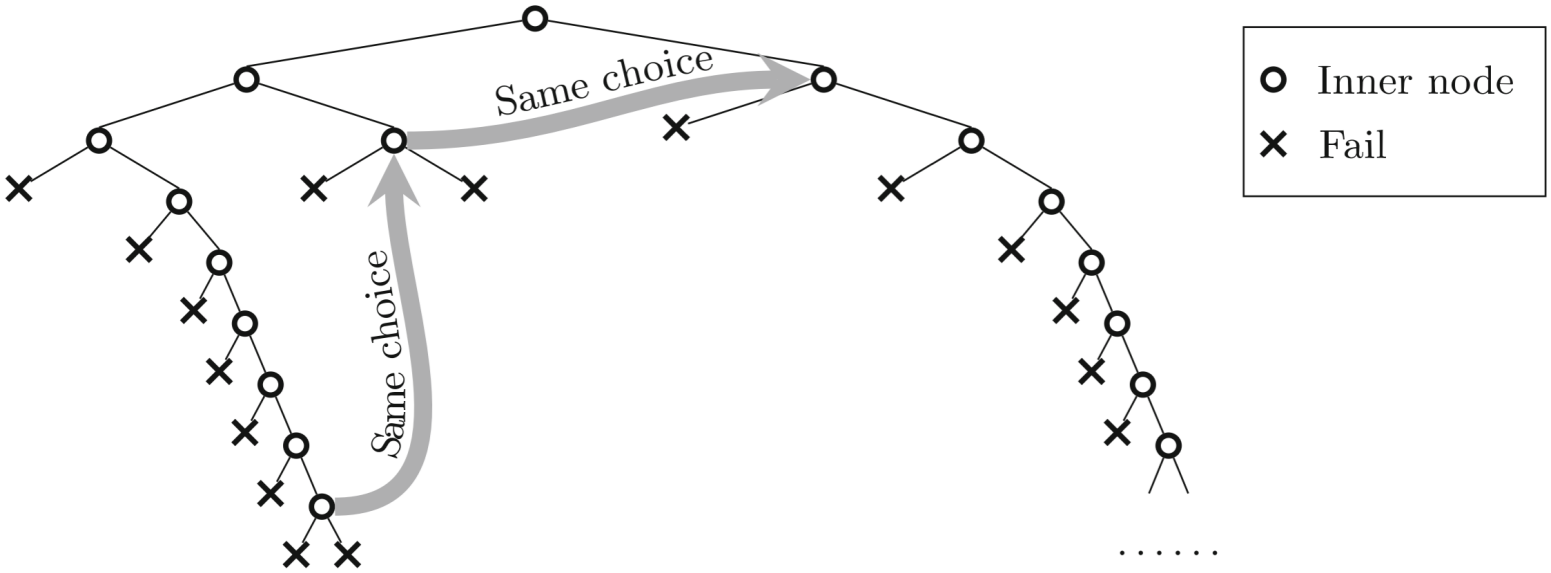
FDS: Branch and Decision Selection



A typical FDS search tree

- Left-branches tend to fail more often
- Decisions that lead to double-fails are re-used frequently

FDS: Branch and Decision Selection



Side-effect: **FDS works well with restarts**

- At each attempt we identify good decisions
- At the next attempt, the search tree will likely be smaller!

Some Results

The paper has an experimentation on scheduling problems:

- Only very hard instances (unknown optimum)
- Objective: minimize a cost metric

Goals of the experimentation:

- Goal 1: prove optimality ("close" the instance)
- Goal 2: improve the best known lower bound
- Goal 3: improve the best known solution (upper bound)

Some Results

Benchmark set	Number of instances	Lower bound improvements	Upper bound improvements	Closed instances
JobShop	48	40	3	15
JobShopOperators	222	107	215	208
FlexibleJobShop	107	67	39	74
RCPSP	472	52	1	0
RCPSPMax	58	51	23	1
MultiModeRCPSP (j30)	552	No reference	3	535
MultiModeRCPSPMax	85	84	77	85

- The results are very good
- Despite the problems are very diverse!

General search strategies are still a very active research topic

- Better results may come in the future!