Writing a CSP Solver in 3 (or 4) Easy Lessons

Christopher Jefferson
University of Oxford

Why Should You Care?

- Understanding how the solver works makes better models.
- Minion already imposes this on you, through its low-level input language.
  - Sorry!

Why Should You Care?

- Often have to go "under the hood" and add new search methods and constraints.
- Don't repeat old mistakes if you write your own constraint solver!

Talk Aims

- These talks will teach you about how constraint solvers work.
- In particular, how Minion works.
- There is not one true way, so my biases will show through!

Talk Aims

- Will discuss GeCode on occasion, as they have nice descriptions of their algorithms.
- Other solvers are interesting, but I don't know what they do.
- Of course mistakes are mine!

Not Talk Aims

- Starting at the Minion Input Language
- Not considering:
  - Tailor
  - ESSENCE'
  - Flattening, Reformulating, ...

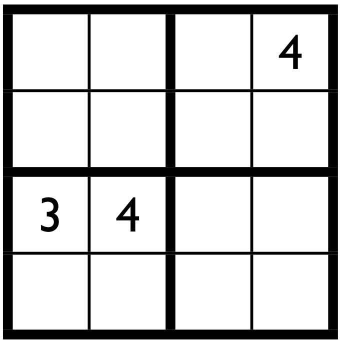
Not Talk Aims

- Not going to teach you how to add new:
  - Variable / Value orders
  - Constraints
  - Variable Types
- But after this talk, its mostly C++
- Happy to talk about this in the lab!

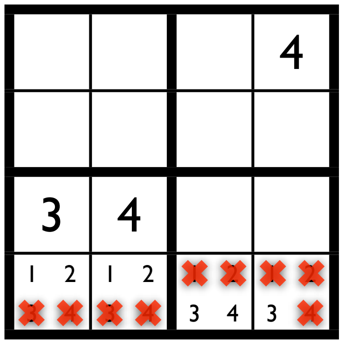
An Aside

Everyone's favourite CSP!

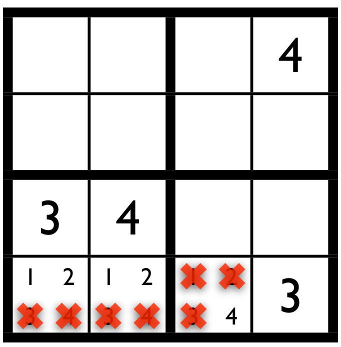
2x2 Sudoku



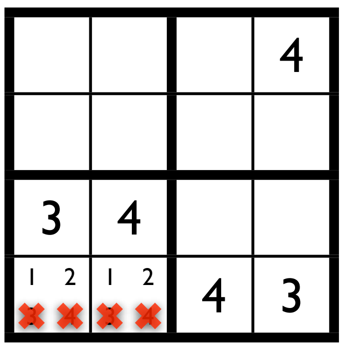
2x2 Sudoku



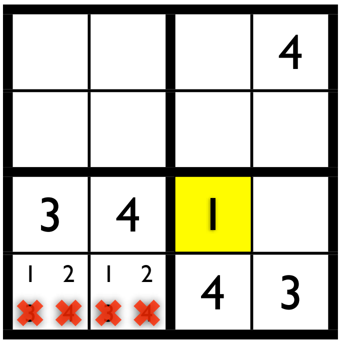
2x2 Sudoku



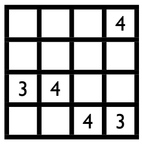
2x2 Sudoku



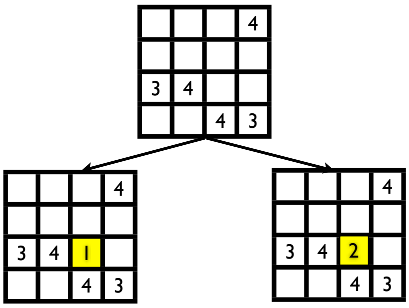
What Now?



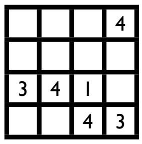
Branching



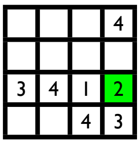
Branching



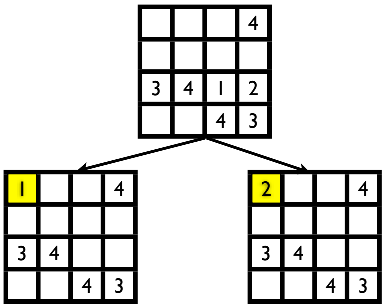
Branching



Branching



Branching



That's It!

- Reduce domains by reasoning with constraints.
- Branch when stuck.

Stupid Solving

Variables	Domain	Time
6	6	~0.5 sec
12	6	~4 hours
12	12	~1,000 years

Assume 100,000 nodes per second

Why is this hard?

- Represent domains.
- Reduce domains with constraints.
- Systematically get all possible reasoning
- Branch and backtrack
- Do it fast!

What is a CSP?

<V, D, C>

What is a CSP Really?

- What is a
  - Domain?
  - Constraint?

The 'Good Old Days'

- Men were men.
- Women were women.
- Constraints were a list of tuples.

X	Y
1	1
2	3
3	4
3	6

Tuples are Well Studied!

AC 1 2 3 4 5 6 7 3.1 2001 3.2 3.3

GAC 4 Schema 2001

Forward Checking

Path Consistency

Directed Arc Consistency

K-consistency

Strong K-consistency

Tuples are Well Studied!

AC 1 2 3 4 5 6 7 3.1 2001 3.2 3.3

GAC 4 Schema 2001

Forward Checking

Path Consistency

Directed Arc Consistency

K-consistency

Strong K-consistency

Propagators

- How constraints are represented inside a constraint solver.
- A black box.
- Offers one operation:
  - Take list of domains, reduce them.

X	1	2	3	4	5	6
Y	1	2	3	4	5	6

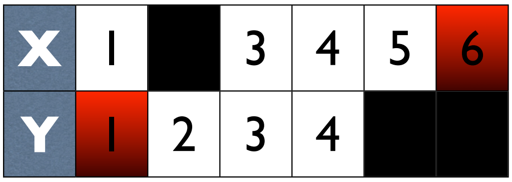
X < Y

X	1	2	3	4	5	6
Y	1	2	3	4	5	6

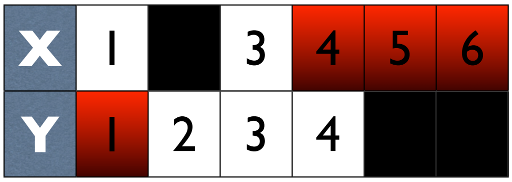
X < Y

X	1	[redacted]	3	4	5	6
Y	1	2	3	4	5	6

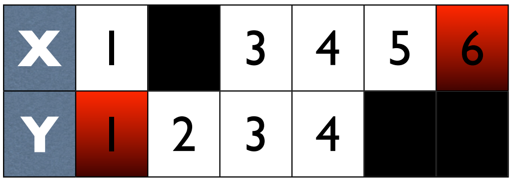
X < Y



$X < Y$



$X < Y$



$X < Y$

Variable Store

- Store a sub-domain for every variable.
- Constraints query and reduce.
- The standard way most CP (and SAT) solvers store state during search.

Variable Store

Given a CSP with variables $\langle V_1,\ldots,V_n \rangle$,
a **search state** is a list of sub-domains:

$$S = \langle D_1,\ldots,D_n \rangle$$

Where D_i is a sub-domain of V_i

Variable Store

- Define a partial order on search states:

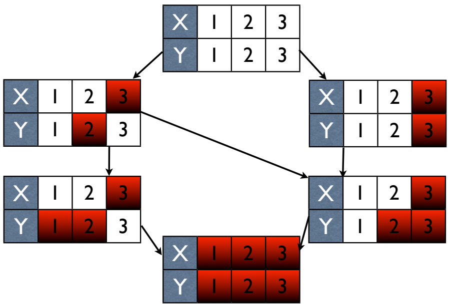
$$S = \langle D_1,\ldots,D_n \rangle,\ \ S' = \langle D_1',\ldots,D_n' \rangle$$

$$S \leq S'$$

$$\updownarrow$$

$$D_1 \subseteq D_1' \,, \ldots , D_n \subseteq D_n'$$

Example Variable Stores



Variable Store

- Set of states form a **lattice**.
  - Lots of nice mathematical results.
- Propagators are a **function** from states to states.
  - S, S' - some states.
  - P - a propagator as a function.

Sensible Requirements

- Do not remove solutions:
  assignment ∈ S → assignment ∈ P(S)
  - Hardest part to get right!

Sensible Requirements

- Monotonic (does not add back domain values):
  S ≤ P(S)
  - This is usually maintained by the solver!

Optional Requirements

- GAC
  - Every domain value left is part of a solution.
  - Or: Strongest valid propagator.
- Easy to show this is well-defined!

Bounds Consistency

- BC (Bounds Consistency)
  - Only check bounds
  - Sortof…
  - Every constraint seems to have a different definition!

Simplest Algorithm

Apply all Propagators

If Any Domain was reduced, repeat

Algorithm Properties

- Fixed point will be reached in a finite amount of time.
  - Assuming finite domains!
  - Infinite domains are scary.
- Fixed point may vary depending on order constraints are executed in.

Standard Requirements

- Confluent: $S \leq S' \rightarrow P(S) \leq P(S')$
  - Lattice Theorem: Whatever order confluent propagators are applied in, same fixed point is reached.

Propagators in Practice

- Lack of confluence is a pain.
  - Reordering propagators can lead to different sized searches.
- But it still gets the right solutions!

The 'Missing Requirement'

- Identifies Solutions:
  If only one value is left in the sub-domain of each variable, reject if not a solution.
- Without this:
  - Need an extra pass at the end of search to check every constraint.
  - "Do Nothing" is a valid propagator.

Improving Propagation

- Two main areas:
  - Reduce how often propagators are run.
  - Speed up propagators.

Constraint Queue

A < B B < C C < D

A < C A < D C < E

Constraint Queue

A < B B < C C < D

A < C A < D C < E

Change A

Constraint Queue

A < B B < C C < D

A < C A < D C < E

Propagate A < B

Constraint Queue

A < B B < C C < D

A < C A < D C < E

Propagate A < D

Change A and D

Constraint Queue

A < B B < C C < D

A < C A < D C < E

Change A and D

Should we re-add A < D to the queue?

Ordering the Queue

- What order should we do things in?
- In theory, it doesn't matter.
- But in practice it does.
  - There is not yet a 'One True Way'.

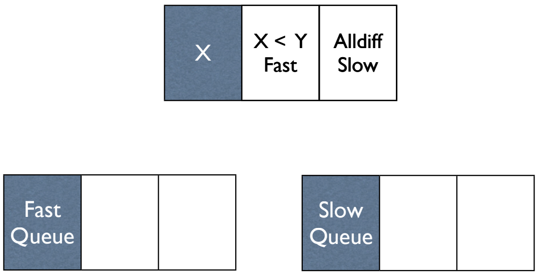
FIFO vs LIFO

- 'First In First Out' faster than 'Last In First Out'
  - Can be faster by magnitudes!
- Further tuning offers much smaller gains.

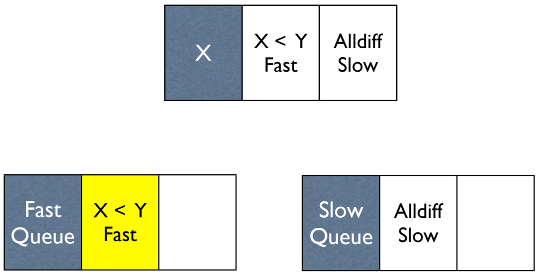
Multiple Queues

- Run the faster things first!
- Gecode has 5 queues.
- Minion has 2 queues.

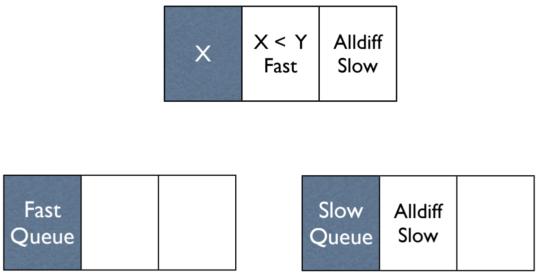
Who Runs the Queues? - GeCode



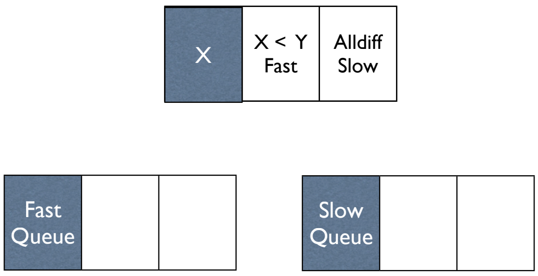
Who Runs the Queues? - GeCode



Who Runs the Queues? - GeCode



Who Runs the Queues? - Minion

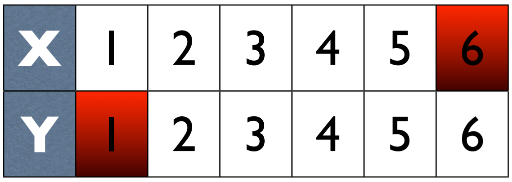


Minion Queues

- Avoids copying queues.
- Queues are precalculated, allocated and compressed before search.
- Faster, but can't be changed.
- Constraints can put themselves on the 'slow queue'.
  - AllDiff, gcc, reification

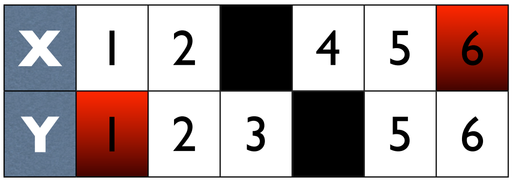
Improving the Queue

- Sometimes we don't care if a variable has changed.
- Allow finer-grained events.



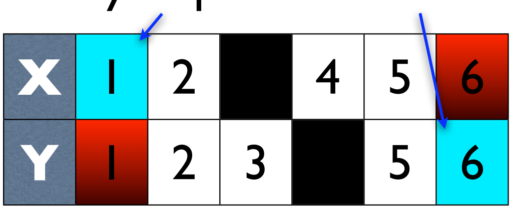
X < Y

Don't Care!



X < Y

Only Important Values



X < Y

Optimising Propagation

- Let constraints state they only want to know about:
  - Lower / Upper Bound.
  - Assignment.
  - Particular Domain Value.
  - Any Change.

Queue

- What exactly goes on the queue?
  - Changed Variables?
  - Changed Constraints?
  - Variable / Constraint pairs?

Other Optimisations

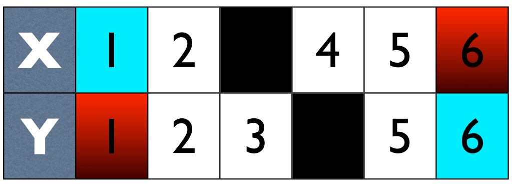
- Merge events.
  - Minion does not try.
- The 'double call problem'.

'Double Call Problem'

- When a variable changes, all the constraints on that variable are added to the queue.
- Including the constraint which just changed the variable!
- It is a pain to get rid of these extra events.
- Minion ignores, GeCode doesn't.

Practical Constraints

Minion's Implementation of X < Y



X < Y

Implementing x≤y

```
LeqConstraint(Var x, Var y)

setupConstraint()
{
  addTrigger(0, x, LowerBound)
  addTrigger(1, y, UpperBound)
}
```

Implementing x≤y

```
LeqConstraint(Var x, Var y)

propagateConstraint(int trigger)
{
  if(trigger == 0)
    y.setMin(x.setMin() + 1)
  else
    x.setMax(y.getMax() - 1)
}
```

Congratulations!

- Our solver now supports the optimal < constraint!
- But, this is not the whole story...

Sum Constraints

- Well researched area.
- We will consider a special case here.
  - Only sum variables of domain {0,1}.
  - Only sum to a constant.

0/1 Sum

- $b_1 + b_2 + \ldots + b_n = c$
  - b_i have domain {0,1}.
  - c is constant.
- This contains most of the ideas of Minion's full sum.

Basic Propagator

- Split variables into 3 sets:
  - $S_0 = \{i \mid Domain(b_i) = \{0\}\}$
  - $S_1 = \{i \mid Domain(b_i) = \{1\}\}$
  - $S_{01} = \{i \mid Domain(b_i) = \{0,1\}\}$
- Eventual sum is between $|S_1|$ and $|S_1| + |S_{01}|$

Basic Propagator

- Split variables into 3 sets:
  - $S_0 = \{i \mid Domain(b_i) = \{0\}\}$
  - $S_1 = \{i \mid Domain(b_i) = \{1\}\}$
  - $S_{01} = \{i \mid Domain(b_i) = \{0,1\}\}$

$c < S_1$	Fail

Basic Propagator

- Split variables into 3 sets:
  - $S_0 = \{i \mid Domain(b_i) = \{0\}\}$
  - $S_1 = \{i \mid Domain(b_i) = \{1\}\}$
  - $S_{01} = \{i \mid Domain(b_i) = \{0,1\}\}$

$c = S_1$	Everything in S_{01} is 0!

Case Split:

$c < S_1$	Fail
$c = S_1$	Everything in S_{01} is 0!
$c = S_1 + S_{01}$	Everything in S_{01} is 1!
$c > S_1 + S_{01}$	Fail

Case Split:

$c < S_1$	Fail
$c = S_1$	Everything in S_{01} is 0!
$S_1 + S_{01} > c > S_1$	???
$c = S_1 + S_{01}$	Everything in S_{01} is 1!
$c > S_1 + S_{01}$	Fail

Not Just Yet...

$S_1 + S_{01} < c < S_1$	???

- Some things in S_{01} must be 1.
  - Some must be 0.
- We don't know which, so we can't propagate.
- We <u>can't</u> just choose some "by symmetry"

Constraint State

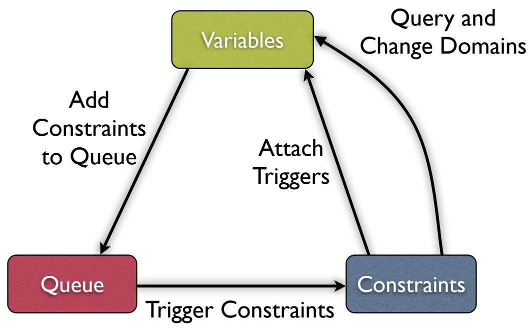
- So far we would have to read the whole array every time a variable changed.
- Can't we just keep a running total which we update?
- Yes we can!

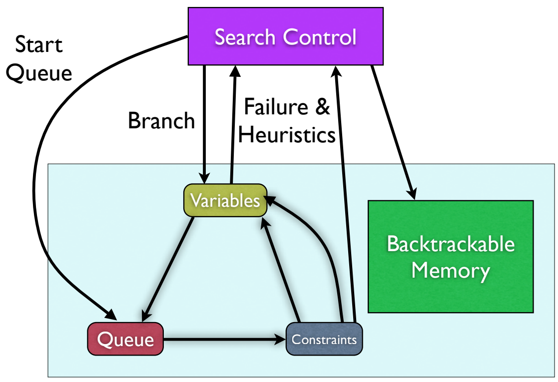
Constraint State

- Allow constraints to store extra state between executions.
- Ensure this is automatically stored on branching and revert on backtrack.
  - Stored just like domains.
- Constraints do not know that branching and backtracking occurs!

Minion

- Reversible<int>
- Acts just like an int in every way.
- Write constraints as if branching never happens, everything "just works".





Variables

The unsung heroes of constraint solvers.

Requirements of Variables

- Good on small domains:
  - Boolean
  - Less than ten.
- Good on huge domains:
  - Thousands or even millions.

Requirements

- Would like it to be fast to:
  - Check and remove values
  - Check and remove ranges.
  - Check and change bounds.
- Fast both in 'O()' and real sense.

It Can't Be Done!

- Minion takes a different route to previous solvers.
  - Provides different implementations of variables, and lets users (or tools) make the choice.
  - Minion doesn't provide the best variable for every situation!

Boolean Variables

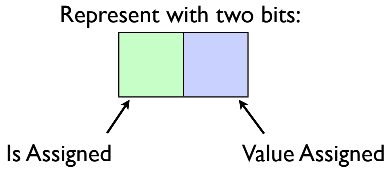
- The very simplest kind of variable.
- But problems instances can contain hundreds of thousands.
- So a well-tuned version is worth putting some work into.

Clever Booleans

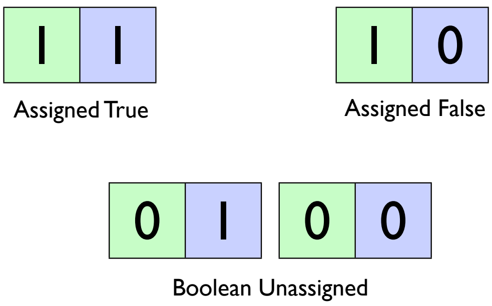
- Three sub-domains:
  - {0,1}, {1}, {0}
- Can be done easily with 2 bits.
- Can we do it with '1.5' bits?
  - Yes, but it's a real pain!

Boolean Variables

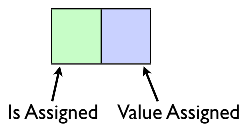
- We can do even better than this!



Boolean Variables



Boolean Variables

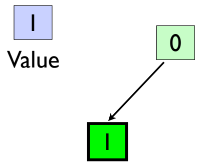


- The "assignment" bit is not backtracked!
  - If variable still assigned, has same value.
  - If unassigned, value unused.

Booleans in Search

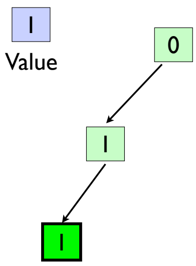
- Start of search.
- "Value" is set to a random value.
- "Assigned" = 0

Booleans in Search



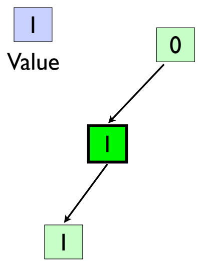
- Search branches.
- The variable is assigned '1'.
- Value and Assigned bits both set.

Booleans in Search



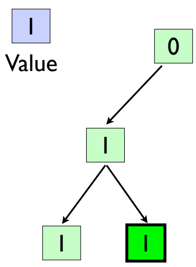
- Search branches.
- The variable is assigned '1'.
- Value and Assigned bits both set.
- Under this node, search leaves variable the same.

Booleans in Search



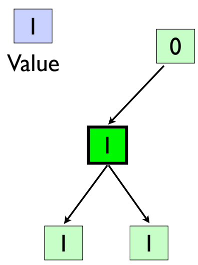
- Search branches.
- The variable is assigned '1'.
- Value and Assigned bits both set.
- Under this node, search leaves variable the same.

Booleans in Search



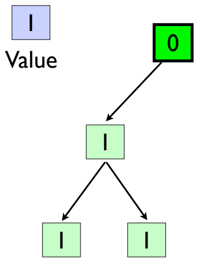
- Search branches.
- The variable is assigned '1'.
- Value and Assigned bits both set.
- Under this node, search leaves variable the same.

Booleans in Search



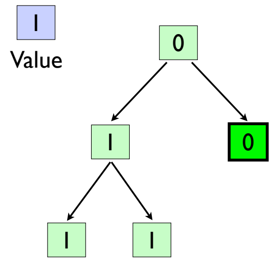
- Search branches.
- The variable is assigned '1'.
- Value and Assigned bits both set.
- Under this node, search leaves variable the same.

Booleans in Search

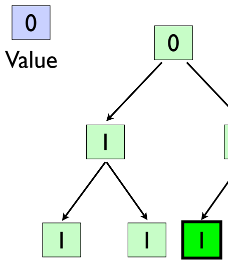


- On backtrack here, "Assigned" set to 0.
- "Value" will be ignored until Boolean is next assigned.

Booleans in Search

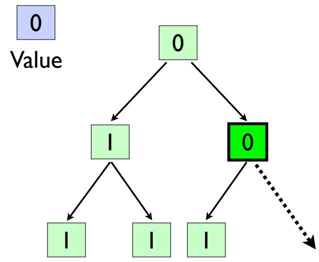


Booleans in Search



- Variable assigned 0.
- "Assigned" set to 1.
- "Value" set to 0.

Booleans in Search



- "Assigned" reset on backtrack
- "Value" left alone.
- Search continues...

Non-Backtracked Data Structures

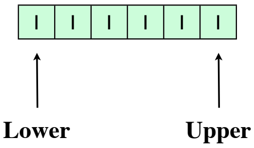
- The 'assigned' value can change on backtrack, but the value is still correct.
- Many such data structures in SAT.
- Becoming increasingly popular in CP (or at least in Minion!)
- Proofs of correctness (and bugs in them) can be very subtle.

Inside a Boolean Variable

- A Boolean Variable is:
  ```
  int* assignPtr
  int* valPtr
  int mask
  ```
- Makes checking / assigning very quick
  - checkAssigned:
    return *assignPtr & mask
  - assignTrue:
    *assignPtr |= mask; *valPtr |= mask

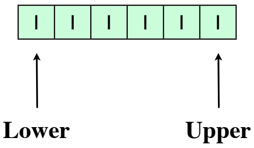
Discrete Variables

- Use a Boolean array for domain values.
- Store upper and lower bounds for optimisation reasons.



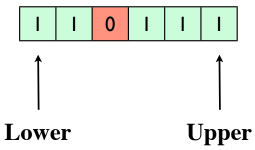
Discrete Variables

- Domain {1,2,3,4,5,6}



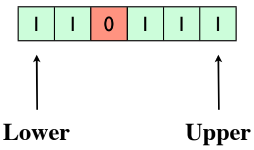
Discrete Variables

- Remove 3 from Domain.



Discrete Variables

- Update **Lower** to 3

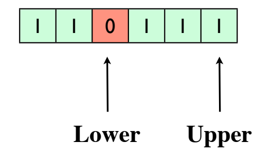


Discrete Variables

- Update **Lower** to 3
- Optimisation:

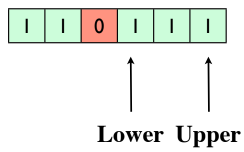
  Boolean array only valid between **Lower** and **Upper**.

  - Gives fast bounds update.



Discrete Variables

- Have to move Lower until the first value in Domain is found.



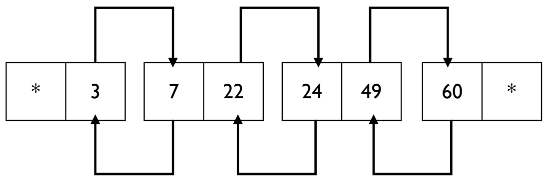
Discrete Variables

- Tweaks can make big differences.
- Add 'cache bounds' to Choco provided a 4 times speed-up on n-queens.

Heavy Duty Variables

- The one model Minion doesn't have is the model most other solvers use!

List of Ranges



Comparison

- For variables of domain < 256
- Bit array:
  - (2 + length/8) bytes - 34 bytes for biggest
- List of ranges
  - Starts with 12 bytes (4 * 4 pointers)
  - Worst case ~ 170 bytes

Constants Matter!

- Don't use bit arrays for huge domains.
- Minion does not handle backtrackable variable-sized allocations. ☹

Large Variables

- Variable of domain {*1..n*}.
- There are 2^n subsets of domain.
- Need n bits to represent in the worst case.

Bound Variables

- Store only the upper and lower bounds.
- Loss of information
  - {1,3,5} → [1 .. 5] → {1,2,3,4,5}
- In Minion, we simply forbid constraints from "poking holes" in the domain.

Bound Variables

- Very small memory usage!
- All operations are very quick!
- Bigger searches.
- Some constraints need special propagators.
  - But not all.

Binary Representation

- We could turn an integer into an array of booleans, under binary representation.
  - 7 = 101
- Takes O(log n) space!
- Is incredibly terrible in almost every case!

Set Variables

- I'm not going to discuss this here.
- Similar basic ideas.
- Can break down into integer variables.
  - Ian Miguel's talk.

Variable Mappers

The Implementer's Secret Code Reduction Trick.

Variable Mappers

X = -Y

Domain of X:
{-3,-1,1,2,10}

Domain of Y:
{3,1,-1,-2,-10}

Variable Mappers

- Consider you want X = -Y.
- Given X's internal state, Y's is redundant.
- Provide "Variable Mappers"
  - Don't store Y's state, just refer to X's

Variable Mappers

- Only store domain once, have other **viewpoints** to it.
- Also need a way of mapping triggers.
  - UpperBound → LowerBound

Mappers in Constraints

$$2X + 3Y - 7Z = 0$$

$$X'=2X,\ Y'=3Y,\ Z'=7Z,$$
$$X'+Y'+Z'=0$$

Mapper Advantages

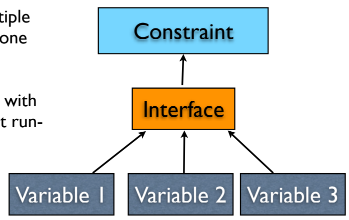
- Can often remove many variables.
- Makes constraints easier to implement.
  - Weighted sum = normal sum + mappers.
    - Imperially as fast as special implementation.
- But mappers are not completely free (division).

Minion Future

- At the moment, mappers are not user-visible.
  - Except Booleans: "!b" means "not b".
- Problems with compile time (more later).

Implementation

- Implementing multiple variable types for one constraint:
- Abstract interface with variables chosen at run-time.
- Slow.
- No inlining.

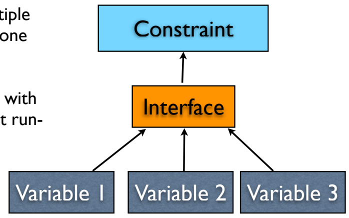


Inside a Boolean Variable

- A Boolean Variable is:
  ```
  int* assignPtr
  int* valPtr
  int mask
  ```
- Makes checking / assigning very quick
  - checkAssigned:
    return *assignPtr & mask
  - assignTrue:
    *assignPtr |= mask; *valPtr |= mask

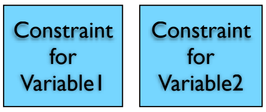
Implementation

- Implementing multiple variable types for one constraint:
- Abstract interface with variables chosen at run-time.
- Slow.
- No inlining.



Different Variable Types

- Implementing multiple variable types for one constraint:
- Implement **each** constraint for **each** type of variable.
- Fast.
- Have to write too much.



Compile-time Interfaces

- Define a minimal interface and **compile** each constraint with each variable type.
- Compiler optimisation removes the interface.
- Allows most constraints to have a single implementation.
- Looking at assembler, often identical to specialised implementations.

Memory Management

Backtracking

- Need to:
  - Store state
  - Revert to an old state when backtracking.
- Encapsulate as much as possible.
  - Constraints should not know about backtracking.

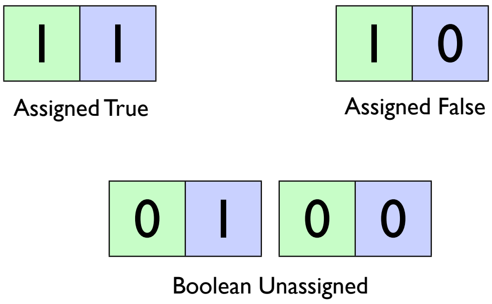
Backtracking

- Trailing
- Copying
- Recomputation

Trailing

- Keep a log of changes made.
- On backtrack, use log to put things back how they were.

Boolean Variables



Trailing Domains

- In Booleans, only ever go from 0 to 1.
- For domain values, only ever remove.
  - So only ever go from 1 to 0.

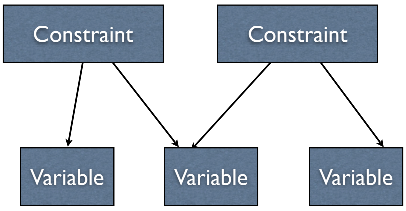
Monotonic Booleans

- Values which only change once during search.
- In this case, only need to know the object changed, we know what it changed to!
- Everything can only change once.

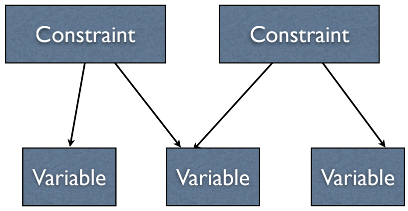
Copying

- Just copy the entire state and restore on backtrack.
- Good for problems with a small state.
  - Many problems do!

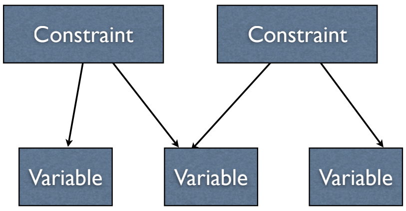
Memory Allocation



Memory Allocation



Memory Allocation



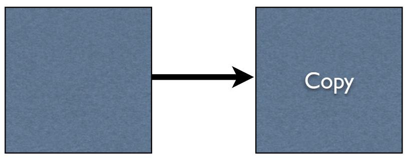
Memory Management

- Choco / GeCode
  - Store a CSP as a tree of objects, explore it to copy.
- Minion
  - Stick everything in fixed memory block at the start, do a "stupid" copy of the memory.

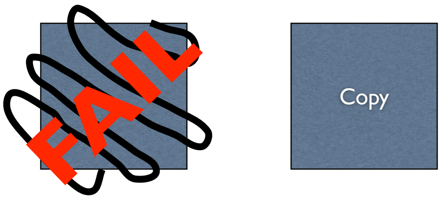
Static vs Dynamic

- Static memory allocation:
  - Does not allow objects to change size.
  - Is much faster to copy at each node.
- Computers are VERY fast at coping blocks of memory.

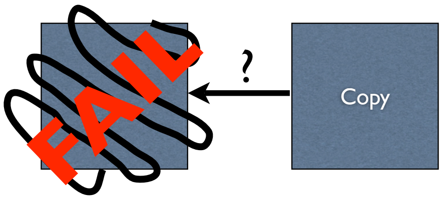
Memory Copying



Memory Copying



Memory Copying



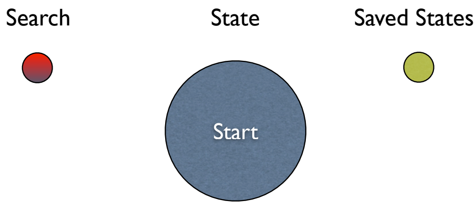
Two Choices

- Have a "Master Place" for state to live, and "saved" copies.
  - Requires more copies.
- Allow "active state" to move around.
  - Extra redirection is expensive.
- Depends on the problem.

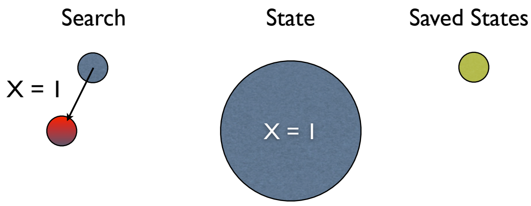
Recomputation

- Regardless of your method, for large problems memory problems get painful.
- Only store state occasionally, and recompute to get back to where you want to be.

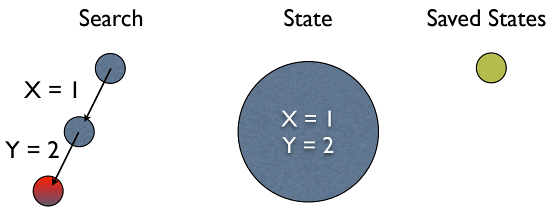
Recomputation Example



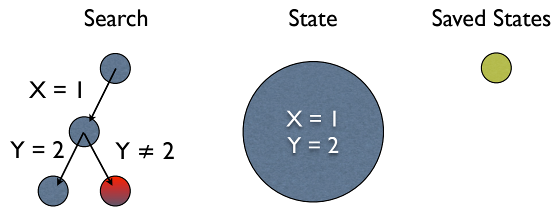
Recomputation Example



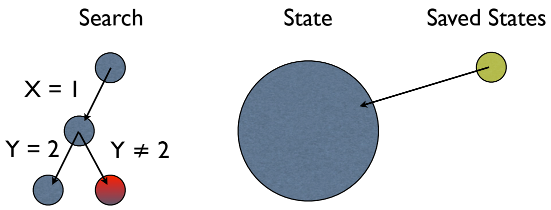
Recomputation Example



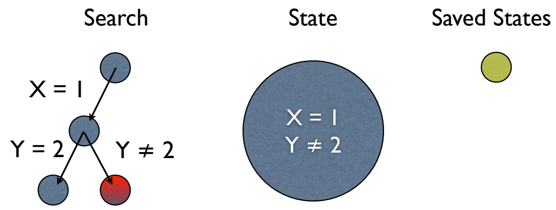
Recomputation Example



Recomputation Example



Recomputation Example



Recomputation

- Need to do extra work when recomputing.
- But save memory!
- Trade-off is not too hard to measure on the fly.
- You can do some other clever things in a recomputation framework - see GeCode

THE END