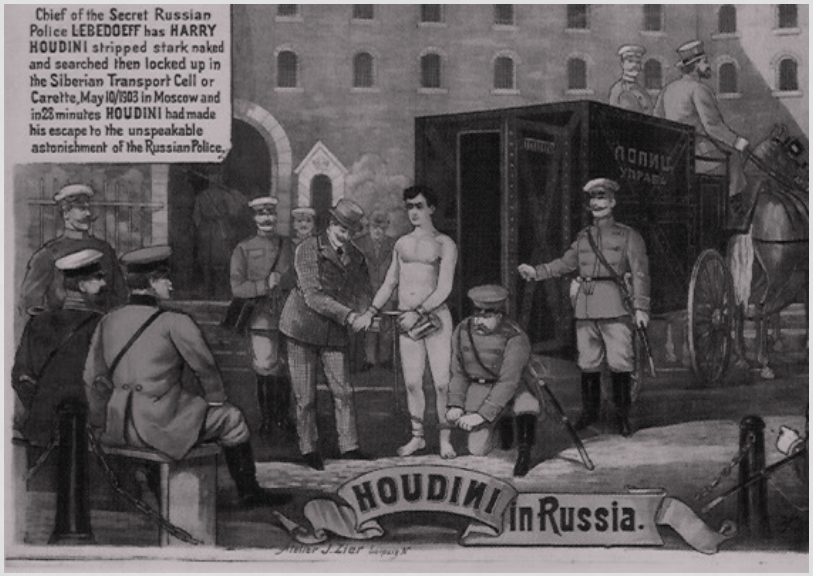


CONSTRAINT-BASED PROBLEM SOLVING

BASIC IDEA

- We don't know how to solve a problem outright.
- We do know rules that it *shouldn't* break when solved
  - The **constraints**.
  - The reverse of a heuristic?
- Usually involves some iteration
  - Try a guess.
  - See what constraints are violated => fix them
  - Repeat.

A CLASSIC LOGIC PROBLEM: **SAT**

- Suppose we have a set of variables:
  - A – E for example.
  - Each can be True or False (the domain)
  - Can be combined with these operators:
    - $\wedge$ $logical - AND$
    - $\vee$ $logical - OR$
    - $\neg$ $logical - NOT$
- SAT is a test to determine (at least one) combination of values to assign to the variables.
- e.g. $((\neg A \vee D) \wedge \neg(C \vee D) \wedge E)$
- This is an NP-hard problem in general

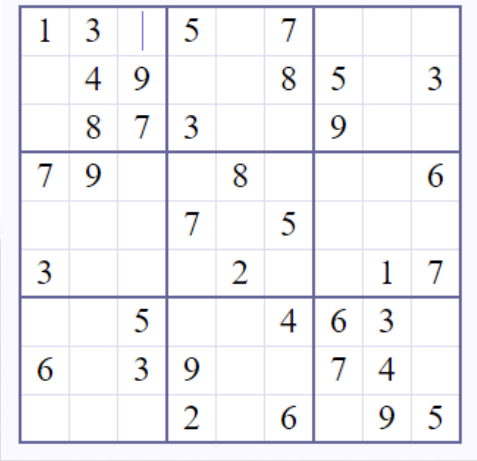
CONNECTION: PROLOG

- Solving this type of problem is what Prolog is built for.
- More general than SAT
  - variables can have any values.
- A few other constraint-based libraries / languages:
  - python-constraint (python package)
  - Google's OR-tools (python, C++, Java library)
  - Screamer (Lisp library)
  - Wolfram (?)

SOME COMMON CONSTRAINTS

- …of a **general** constraint-solver
  - ALL_UNIQUE(list)
  - ALL_DIFFERENT(list)
  - AT_LEAST_ONE(list, val)
  - …
- A huge goal is to avoid **back-tracking**
  - i.e. don't do brute-force.

OUR PROBLEM



1	3		5		7			
	4	9			8	5		3
	8	7	3			9		
7	9			8				6
			7		5			
3				2			1	7
		5			4	6	3	
6		3	9			7	4	
			2		6		9	5

- Solve a Sudko puzzle
- [rules of Sudoku?]
- Have a set of variables for the missing spots.
  - Initially the domain is 1, 2, 3, …, 9
  - if that value would vilolate the Sudoku rules, remove it from the domain for that variable.
  - if you get down to just one value, you know that's the only answer. Use that to eliminate other values from other variables domains.
  - for easy – medium puzzles, repeatedly applying should let you solve.
  - for harder puzzles, you may have to do *some* trial-and-error (backtracking)

A RELATED (?) PROBLEM: VERLET PHYSICS

- *Might* be a stretch…but it's cool
- https://www.youtube.com/watch?v=w88D8-0-Kb4
- https://www.youtube.com/watch?v=DWTpjNUbT24
- Discovered by Hitman: Codename 47 (2000)
  - http://graphics.cs.cmu.edu/nsp/course/15-869/2006/papers/jakobsen.htm
- Basic idea:
  1. a set of points
  2. a set of constraints between points.
  3. apply gravity and mouse-drag to points
  4. iteratively fix the constraints
  5. soft-body physics! ragdoll! cloth-simulators!
  6. profit!

POSITION-INTEGRATION

- What we usually use (Euler integration):
  - class PointMass{ Position p; Velocity v; }
    - def update(dt, a):
      - v += a * dt
      - p += v * dt
- Verlet Integration
  - Don't store velocity!
  - Instead store it's last position (and current)
  - change update to:
    - x_new = 2x – x + a * dt * dt (chang 2 to 1.99 or so to do friction)
    - x_old = x
    - x = x_new

CONSTRAINT-SATISFACTION

- Define a constraint as:
  - connecting 2 points (p1, p2)
  - a distance (could calculate from p1, p2 initial values)
  - [optional] springiness
    - how aggressive are we at fixing constraints?
  - [optional] tolerance
    - allow this much variation from rest-distance before correcting.
- Update procedure
  - Every frame (perhaps multiple times?) fix the constraints.
- That's it!
- [demo]