CS486/686: Introduction to Artificial Intelligence
Lecture 4 - Features and Constraints

Jesse Hoey & Victor Zhong

School of Computer Science, University of Waterloo

January 16, 2025

Readings: Poole & Mackworth Chap. 4.1-4.7

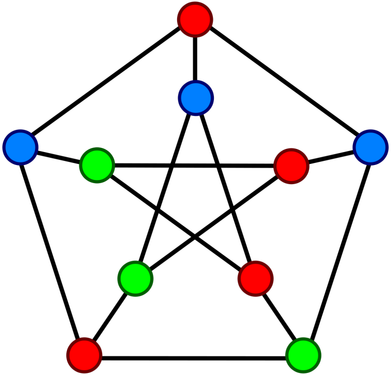
Evacuation Planning

- Instruct residents to follow a route at a given time
- Two challenges:
  (1) deploy enough resources to give instructions
  (2) ensure that the population comply with the instructions

- Applied in a real-life case study and
  generated schedules close to the optimal ones from prior work

- Even, C., Schutt, A., & Van Hentenryck, P. (2015).
  A constraint programming approach for non-preemptive evacuation scheduling. `https://arxiv.org/pdf/1505.02487.pdf`

Graph Coloring Problem

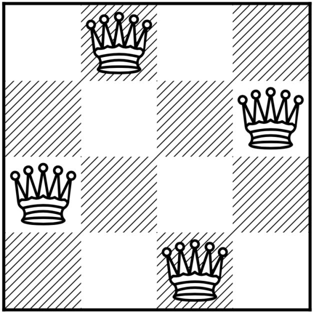


Nodes on the same edge cannot have the same color

Sudoku

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

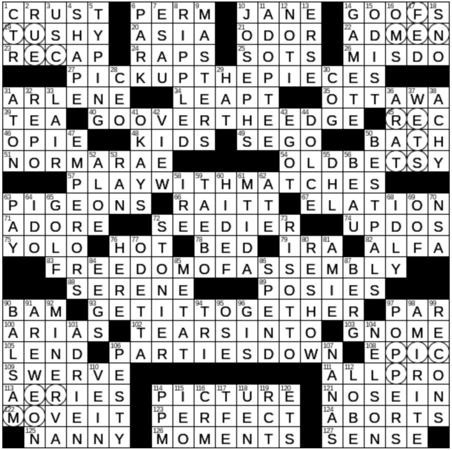
N-Queens Problem



The queens can attack each other if they are in the same row, column, or diagonal

Place 4 queens on a 4x4 chessboard so that there is no attack

Crossword Construction



Fill in all horizontal and vertical slots with words or phrases

Constraint Satisfaction Problems (CSPs)

- A set of **variables**
- A **domain** for each variable
- A set of **constraints** **or** **evaluation function**
- Two kinds of problems:
  1. **Satisfiability** Problems: Find an assignment that satisfies constraints (**hard** constraints)
  2. **Optimization** Problems: Find an assignment that optimises the evaluation function (**soft** constraints)
- A **solution** to a CSP is an assignment to the variables that satisfies all constraints
- A solution is a **model** of the constraints

CSPs as Graph searching problems

Two ways to represent CSPs as graph searching:

Complete Assignment:

- Nodes: assignment of value to all variables
- Neighbors: change one variable value

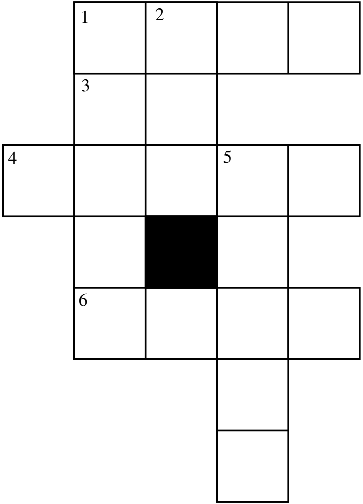
Partial Assignment:

- Nodes: assignment to first $k-1$ variables
- Neighbors: assignment to k^{th} variable

But,

- These search spaces can get extremely large (thousands of variables), so the branching factors can be big
- Path to goal is not important, only the goal is
- No predefined starting nodes

Classic CSP: Crossword Construction



at,eta,be,hat,he,her,it,him

on,one,desk,dance,usage,easy,dove

first,else,loses,fuels,help,haste,

given,kind,sense,soon,sound,this,think

Dual Representations

Two ways to represent the crossword as a CSP

- **Primal** representation:
  - Nodes represent word positions: 1-down…6-across
  - Domains are the words
  - Constraints specify that the letters on the intersections must be the same
- **Dual** representation:
  - Nodes represent the individual squares
  - Domains are the letters
  - Constraints specify that the words must fit

Real World Example Domains

- Transportation Planning (Pascal Van Hentenryck) `https://www.youtube.com/watch?v=SxvM0jG3qLA`
- Ride-sharing scheduling `https://arxiv.org/pdf/2111.03204`
- Air Traffic Control `http://dx.doi.org/10.1017/S0269888912000215`
- Disaster Recovery
- Factory process management
- Scheduling (courses, meetings, etc)
- ...

Posing a CSP

Variables: $V_1, V_2, \ldots, V_n$

Domains: for each variable, V_i has a domain D_{V_i}

Constraints: restrictions on the values a set of variables can jointly have

e.g.

problem	**variables**	**domains**	**constraints**
crosswords	letters	a-z	words in dictionary
crosswords	words	dictionary	letters match
scheduling	times	times,dates	before, after
	events	types	same resource
	resources	values	
party planning	guests	values	cliques
ride sharing	people/trips	locations	cars

Constraints

Constraints:

- Can be **N-ary** (over sets of N variables - e.g. "dual representation" for crossword puzzles with letters as domains)
- Here: Consider only **Unary** and **Binary** (e.g. "primal representation" for crossword puzzles with words as domains)

Solutions:

- Generate and test
- Backtracking
- Consistency
- Hill-Climbing
- Randomized incl. Local Search

Example

Delivery robot: activities a,b,c,d,e, times 1,2,3,4

A: variable representing the time activity a will occur

B: variable representing the time activity b will occur

...

Domains:

$D_A = \{1, 2, 3, 4\}$
$D_B = \{1, 2, 3, 4\}$

...

Constraints :

$$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge$$
$$(C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge (E < C)$$
$$\wedge (E < D) \wedge (B \neq D)$$

Generate and Test

$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge (C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge (E < C) \wedge (E < D) \wedge (B \neq D)$

Exaustively go through all combinations, check each one

$D = D_A \times D_B \times D_C \times D_D \times D_E$

$D = \{< 1, 1, 1, 1, 1 >, < 1, 1, 1, 1, 2 >, \ldots, < 4, 4, 4, 4, 4 >\}$

Test: $< 1, 1, 1, 1, 1 > \ldots$ fail $\neg(A \neq B)$
Test: $< 1, 1, 1, 1, 2 > \ldots$ fail $\neg(A \neq B)$
Test: $< 1, 1, 1, 1, 3 > \ldots$ fail $\neg(A \neq B)$

...

...

Test: $< 1, 2, 1, 1, 1 > \ldots$ fail $\neg(C < D)$
Test: $< 1, 2, 1, 1, 2 > \ldots$ fail $\neg(C < D)$

...

but ... **we knew all along** that $A \neq B$

Backtracking

$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge (C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge (E < C) \wedge (E < D) \wedge (B \neq D)$

Can use the fact that large portions of the state space can be pruned?

1. Order all variables
2. Evaluate constraints into the order as soon as they are grounded

e.g. Assignment $A = 1 \wedge B = 1$ is inconsistent with constraint $A \neq B$ regardless of the value of the other variables.

Backtracking: Example

$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge (C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge (E < C) \wedge (E < D) \wedge (B \neq D)$

Test: $< 1, \_, \_, \_, \_ >$ ... ok
Test: $< 1, 1, \_, \_, \_ >$ ... fail $\neg(A \neq B)$
Test: $< 1, 2, 1, \_, \_ >$ ... ok
Test: $< 1, 2, 1, 1, \_ >$ ... fail $\neg(C < D)$
Test: $< 1, 2, 1, 2, \_ >$ ... fail $\neg(A = D)$
Test: $< 1, 2, 1, 3, \_ >$ ... fail $\neg(A = D)$
Test: $< 1, 2, 1, 4, \_ >$ ... fail $\neg(A = D)$
Backtrack
Test: $< 1, 2, 2, \_, \_ >$ ... fail $\neg(C \neq 2)$
Test: $< 1, 2, 3, \_, \_ >$ ... ok
Test $< 1, 2, 3, 1, \_ >$ ... fail $\neg(C < D)$
...
(draw the search tree using the partial assignment method)

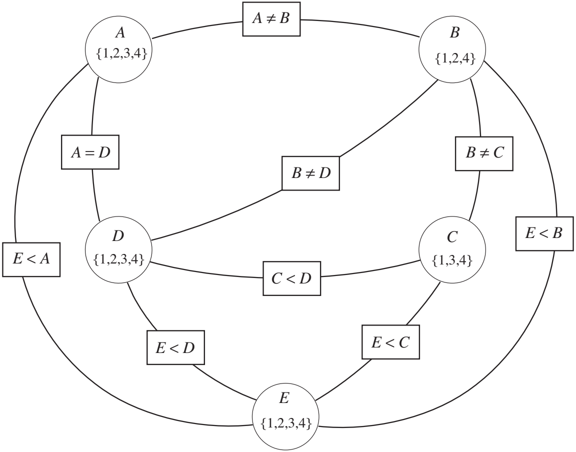
Backtracking

- Efficiency depends on **order** of variables
- Finding optimal ordering is **as hard** as solving the problem
- Idea: **push failures** as high as possible
- **Cut off** large branches of the tree as soon as possible

Consistency

- More general approach
- Look for **inconsistencies**
  e.g. C=4 in example inconsistent with any value of D ($C < D$)
- Backtracking will "re-discover" this
- **Graphical** representation

Constraint Satisfaction: Graphically



Goal: each domain takes a value, and all constraints are satisfied

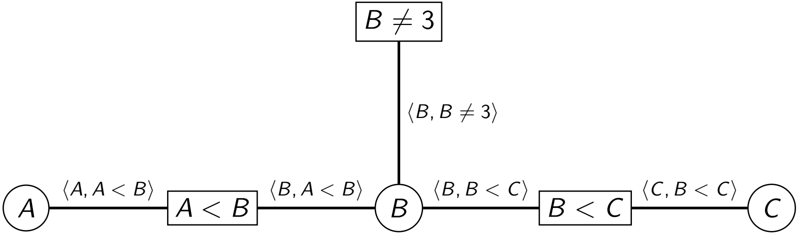
Consistency Networks (CN)

- **Domain constraint** is unary constraint on values in a domain, written $\langle X, c(X) \rangle$
- A node in a CN is **domain consistent** if no domain value violates any domain constraint
- A CN is **domain consistent** if all nodes are **domain consistent**

- Arc $\langle X, c(X, Y) \rangle$ is a **constraint** on X.
- An arc $\langle X, c(X, Y) \rangle$ is **arc consistent** if for each $X \in \mathbf{D}_X$, there is some $Y \in \mathbf{D}_Y$ such that $c(X, Y)$ is satisfied.
- A CN is **arc consistent** if all arcs are **arc consistent**

- A set of variables $\{X_1, X_2, X_3, \ldots, X_N\}$ is **path consistent** if all arcs and domains are consistent

Constraint Satisfaction: Graphically (Formal)



AC-3

- Alan Mackworth 1977
- Makes a CN **arc consistent** (and **domain consistent**)
- To-Do Arcs (TDA) Queue has all inconsistent arcs

1.**Make** all domains domain consistent
2.**Put** all arcs $\langle Z, c(Z, \_)\rangle$ in TDA
3.**Repeat**
    a.**Select** and remove an arc $\langle X, c(X, Y)\rangle$ from TDA
    b.**Remove** all values of domain of X
    that don't have a value in domain of Y that satisfies
        the constraint $c(X, Y)$
    c.**If** any were removed,
        Add all arcs $\langle Z, c'(Z, X)\rangle$ to TDA $\forall \mathbf{Z} \neq \mathbf{Y}$
until TDA is empty

When AC-3 Terminates

AC-3 **always** terminates with one of these three conditions:

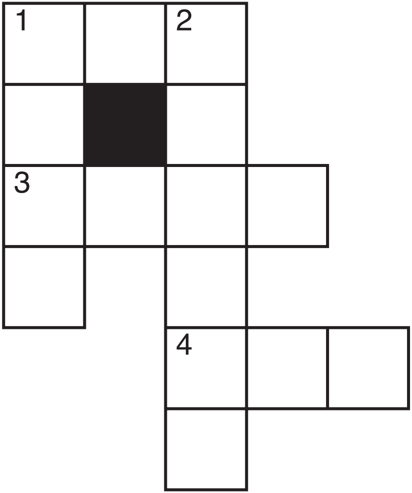
- Every domain is empty: there is **no solution**
- Every domain has a single value: **solution!**
- Some domain has more than one value: split it in two, run AC-3 recursively on two halves
  Don't have to start from scratch - only have to put back all arcs $\langle Z, c'(Z, X)\rangle$ if X was the domain that was split
- Connection between domain splitting and search

Properties of the AC-3 Algorithm

- Is AC-3 guaranteed to terminate?
  Yes.

- What is the complexity of AC-3?
  n variables, c binary constraints, and the size of each domain is at most d
  Time complexity: $O(cd^3)$
  Each arc (X_k, X_i) can be added to the queue at most d times because we can delete at most d values from X_i
  Checking consistency of each arc can be done in $O(d^2)$ time

Example: Crossword Puzzle



Words:

ant, big, bus, car, has
book, buys, hold,
lane, year
beast, ginger, search,
symbol, syntax

Variable Elimination

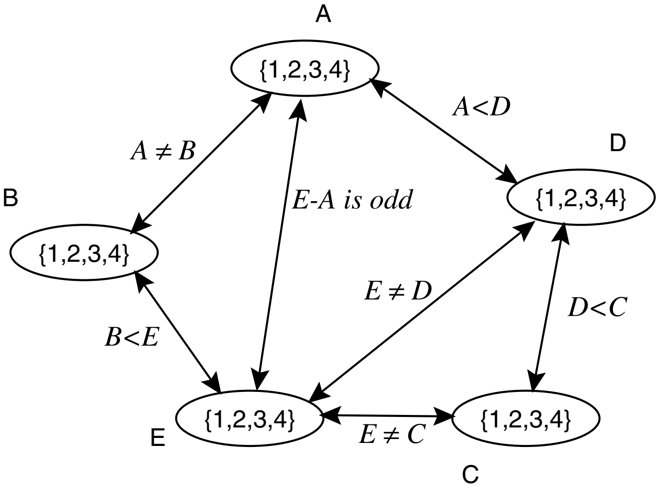
- Idea: **eliminate** the variables one-by-one passing their constraints to their neighbours
- When there is a single variable remaining, if it has no values, the network was **inconsistent**.
- The variables are eliminated according to some **elimination ordering**
- Different elimination orderings result in different size intermediate constraints.

Variable Elimination (cont.)

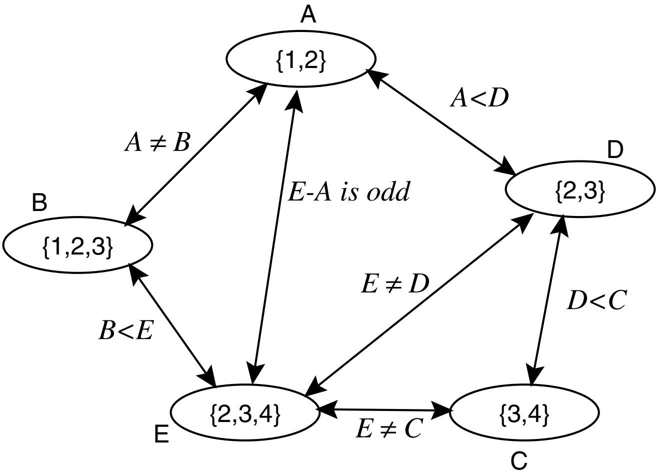
Variable Elimination Algorithm:

- If there is only one variable, return the intersection of the (unary) constraints that contain it
- Select a variable X
  - Join the constraints in which X appears, forming constraint R
  - Project R onto its variables other than X: call this R_2
  - Place new constraint R_2 between all variables that were connected to X
  - Remove X
  - Recursively solve the simplified problem
  - Return R joined with the recursive solution

Example Network



Example: Arc-Consistent Network



Example: Eliminating C

$r_1 : C \neq E$	C	E
	3	2
	3	4
	4	2
	4	3

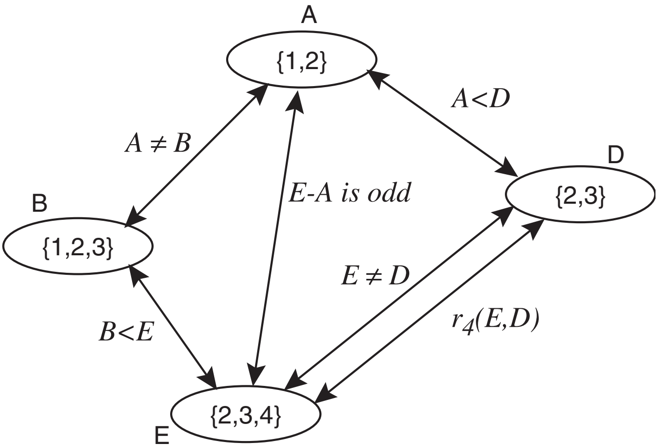
$r_2 : C > D$	C	D
	3	2
	4	2
	4	3

$r_3 : r_1 \bowtie r2$	C	D	E
	3	2	2
	3	2	4
	4	2	2
	4	2	3
	4	3	2
	4	3	3

$r_4 : \pi_{\{D,E\}} r_3$	D	E
	2	2
	2	3
	2	4
	3	2
	3	3

$\hookrightarrow$ new constraint

Resulting Network after Eliminating C



Local Search

Back to CSP as Search (**Local Search**):

- **Maintain** an assignment of a value to each variable
- At each step, select a **neighbor** of the current assignment (e.g., one that improves some **heuristic** value)
- Stop when a **satisfying** assignment is found, or return the best assignment found

Requires:

- What is a neighbor?
- Which neighbor should be selected?

(Some methods maintain **multiple assignments**)

Local Search for CSPs

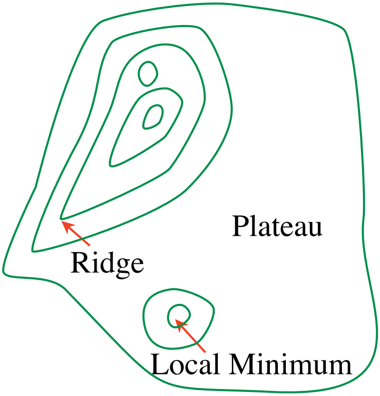
- Aim is to find an assignment with **zero unsatisfied** constraints
- Given an assignment of a value to each variable, a **conflict** is an unsatisfied constraint
- The goal is an assignment with **zero conflicts**
- **Heuristic** function to be minimized: the number of conflicts

Greedy Descent Variants

- Find the variable-value pair that **minimizes** the number of conflicts at every step
- Select a variable that participates in the **most** number of conflicts
  Select a value that **minimizes** the number of conflicts.
- Select a variable that appears in **any** conflict
  Select a value that **minimizes** the number of conflicts.
- Select a variable at **random**
  Select a value that **minimizes** the number of conflicts
- Select a variable **and** value at **random**; **accept** this change if it **doesn't increase** the number of conflicts.

Problems with Greedy Descent

- A **local minimum** that is not a global minimum
- A **plateau** where the heuristic values are uninformative
- A **ridge** is a local minimum where *n*-step look-ahead might help



Randomized Greedy Descent

As well as downward steps we can allow for:

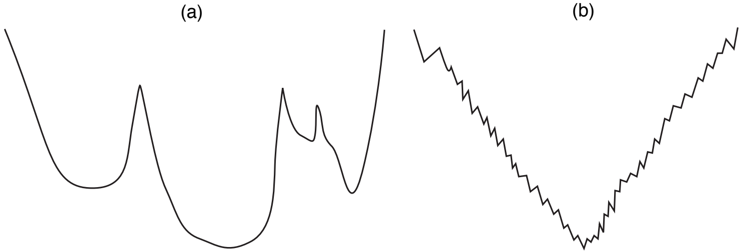
- **Random steps:** move to a random neighbor
- **Random restart:** reassign random values to all variables

Which is more expensive computationally?

A mix of the two = **stochastic local search**

1-Dimensional Ordered Examples

Two 1-dimensional search spaces; step right or left:



- Which method would most easily find the **global** minimum?
- What happens in hundreds or **thousands of dimensions**?
- What if different parts of the search space have **different structure**?

High Dimensional Search Spaces

- In high dimensions the search space is less easy to visualize
- Often consists of long, nearly flat **"canyons"**
- Hard to optimize using local search
- Step-size can be adjusted

Stochastic Local Search

Stochastic local search is a mix of:

- **Greedy descent**: move to a lowest neighbor
- **Random walk**: taking some random steps
- **Random restart**: reassigning values to all variables

Variant: Simulated Annealing

- Pick a variable at random and a new value at random
- If it is an improvement, adopt it
- If it isn't an improvement, adopt it probabilistically depending on a temperature parameter, T
  - With current assignment n and proposed assignment n' we move to n' with probability $e^{-(h(n')-h(n))/T}$
- Temperature can be reduced.

Probability of accepting a change:

Temperature	1-worse	2-worse	3-worse
10	0.91	0.81	0.74
1	0.37	0.14	0.05
0.25	0.02	0.0003	0.000005
0.1	0.00005	0	0

Simulated Annealing

Let n be random assignment of values to all variables
Let T be a (high) temperature
repeat
    Select neighbor n' of n at random
    If $h(n') < h(n)$ then
        $n = n'$
    else
        $n = n'$ with probability $e^{-(h(n')-h(n))/T}$
    reduce T
until stopping criteria is reached

Tabu Lists

- Recall GSAT (Greedy SATisfiability): never choose same variable twice
- To prevent cycling we can maintain a **tabu list** of the k last assignments
- Don't allow an assignment that is already on the tabu list
- If $k = 1$, we don't allow an assignment of to the same value to the variable chosen
- We can implement it more efficiently than as a list of complete assignments
- It can be expensive if k is large

Parallel Search

A total assignment is called an **individual**

- maintain a **population** of k individuals instead of one
- At every stage, **update** each individual in the population
- Whenever an individual is a solution, it can be reported
- Like k restarts, but uses k times the minimum number of steps

Beam Search

- Like parallel search, with k individuals, but choose the k **best** out of all of the neighbors (all if there are less than k)
- When $k = 1$, it is greedy descent
- The value of k lets us limit space and parallelism

Stochastic Beam Search

- Like beam search, but it **probabilistically** chooses the k individuals at the next generation.
- The probability that a neighbor is chosen is proportional to its **heuristic** value : $e^{-h(n)/T}$
- This maintains **diversity** amongst the individuals
- The heuristic value reflects the fitness of the individual
- Like **asexual** reproduction: each individual mutates and the fittest ones survive

Genetic Algorithms

- Like stochastic beam search, but pairs of individuals are combined to create the **offspring**:
- For each generation:
  - Randomly choose pairs of individuals where the fittest individuals are more likely to be chosen
  - For each pair, perform a **cross-over**: form two offspring each taking different parts of their parents:
  - **Mutate** some values
- Stop when a solution is found

Crossover

- Given two individuals:

$$X_1 = a_1, X_2 = a_2, \ldots, X_m = a_m$$

$$X_1 = b_1, X_2 = b_2, \ldots, X_m = b_m$$

- Select i at random
- Form two offsprings:

$$X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = b_{i+1}, \ldots, X_m = b_m$$

$$X_1 = b_1, \ldots, X_i = b_i, X_{i+1} = a_{i+1}, \ldots, X_m = a_m$$

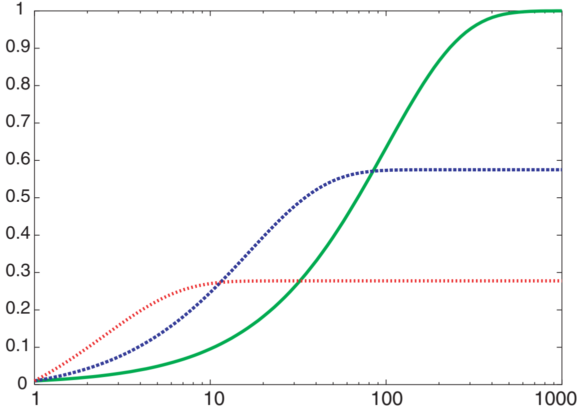
- The effectiveness depends on the ordering of the variables
- Many variations are possible

Comparing Stochastic Algorithms

- How can you **compare** three algorithms when
  - one solves the problem 30% of the time very quickly but doesn't halt for the other 70% of the cases
  - one solves 60% of the cases reasonably quickly but doesn't solve the rest
  - one solves the problem in 100% of the cases, but slowly?
- Summary statistics, such as mean run time, median run time, and mode run time don't make much sense

Runtime Distribution

- Plots runtime (or number of steps) and the proportion (or number) of the runs that are solved within that runtime



Next

- Inference (Poole & Mackworth chapter 5.1-5.3 and 6.1-6.3)