# Local search algorithms

Devika Subramanian

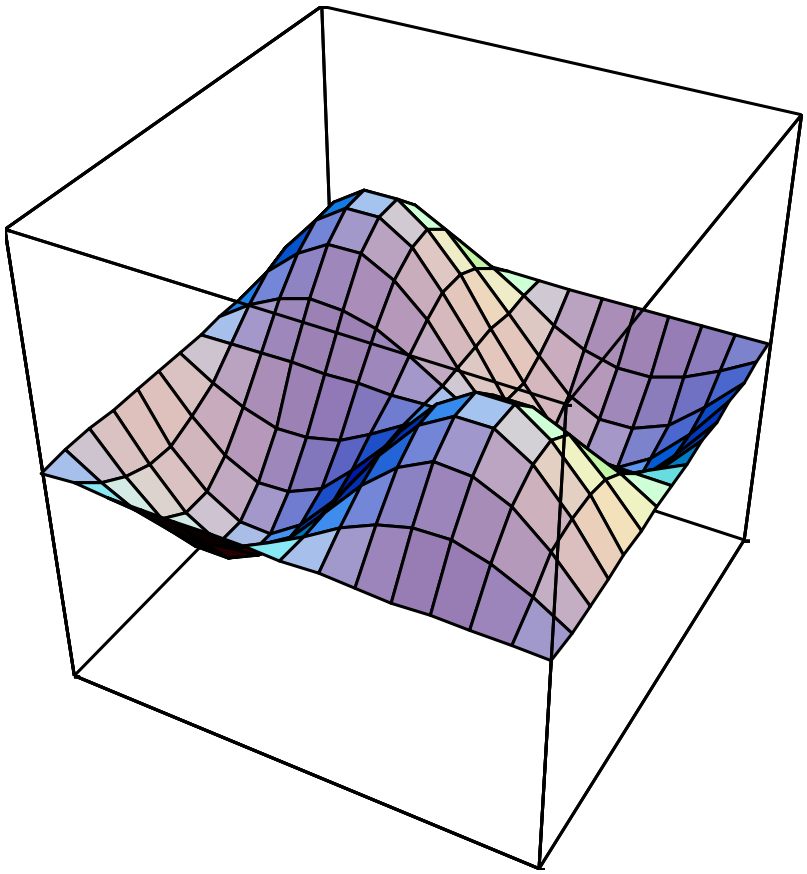
# Local search algorithms

- Used for problems where finding a goal state is more important than finding the shortest path to it. Examples: airline scheduling, channel routing, optical design, VLSI layout, boolean satisfiability, N-queens, protein folding, sequence alignment.
- We will cover three algorithms from this family
  - Hill climbing and two randomized variants
  - Simulated annealing
  - Genetic algorithms

# Informal characterization of problems

- There is a combinatorial structure being optimized (a schedule, a tour, an assignment of truth values to Boolean variables, etc).
- There is a function c: structure → R representing cost or quality of a structure.
- The goal of search is to find a structure with max. quality or min. cost.
- Searching all possible structures is impossible.
- There is no known algorithm for computing the optimal solution efficiently.
- However, similar solutions have similar costs.

# Local search



We have a space of structures or configurations and each has an associated quality.

Optimization problem: The goal is to find the best solution. How we get to it is unimportant.

Decision problem: the space is divided into structures with cost 1 (solutions) and cost 0 (non-solutions).
Goal is to find any solution.

# Recipe for local search algorithms

- 1: Start at a random state.
- 2: Make local modifications (neighbors) to improve the current state.
- 3: Repeat Step 2 until goal found or you run out of time.

Key: designing a set of local modifications or neighbors.

# An example: MAX-3-SAT

- Given a set of 3-clauses (disjunctions of three literals) in the propositional calculus, find an assignment of truth values to the variables which maximizes the number of satisfied clauses.

- Example:

$$A \vee \neg B \vee C$$

$$\neg A \vee C \vee D$$

$$B \vee D \vee \neg E$$

$$\neg C \vee \neg D \vee \neg E$$

$$\neg A \vee \neg C \vee E$$

- Structures are assignments to variables A,B,C,D,E.
- We represent them as 5-tuples over {0,1}.
- For (1,0,1,0,1), how many of the clauses are satisfied?

# Design of neighbors: part 1

- Neighbors: Hamming 1 neighbors.
  - Neighbors of (1,0,1,0,1) are those which differ from it in exactly one bit.

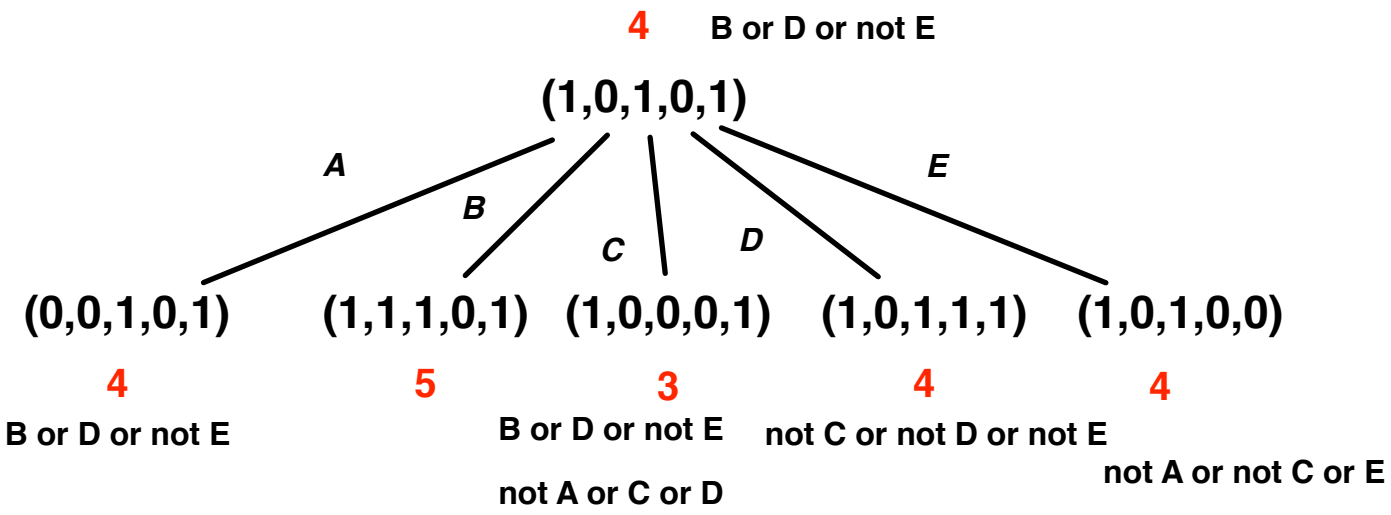
$A \vee \neg B \vee C$

$\neg A \vee C \vee D$

$B \vee D \vee \neg E$

$\neg C \vee \neg D \vee \neg E$

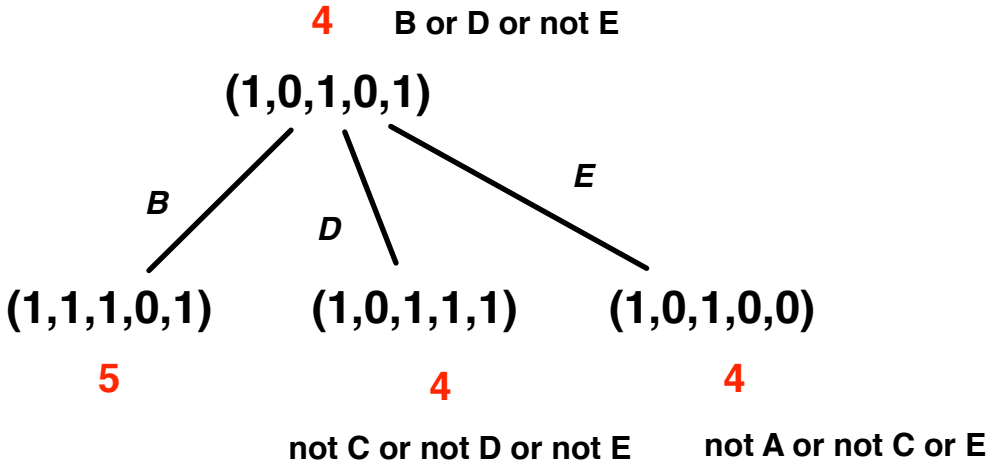
$\neg A \vee \neg C \vee E$




Quality measure = number of satisfied clauses

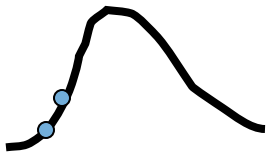
# Design of neighbors: part 2

- Neighbors: flip assignment of a single variable in an unsatisfied clause.
  - Only B, D or E needs to be flipped.



# Hill climbing (steepest ascent)

- function HC (problem) returns a state
  - current = a random state
  - Do forever
    - next = a neighbor of current with the highest value [examine ALL neighbors]
    - If value(next) <= value(current) then return current
    - current = next

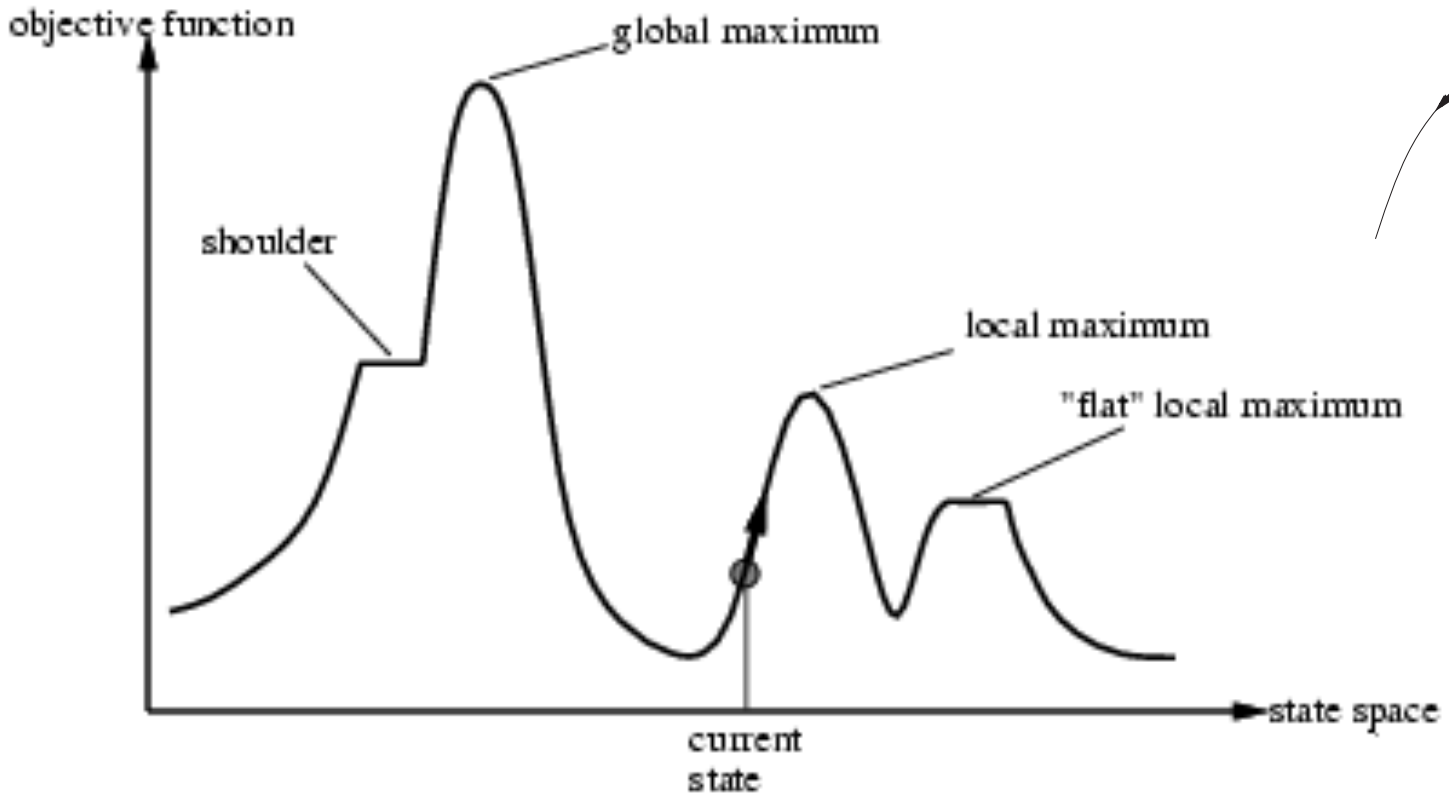


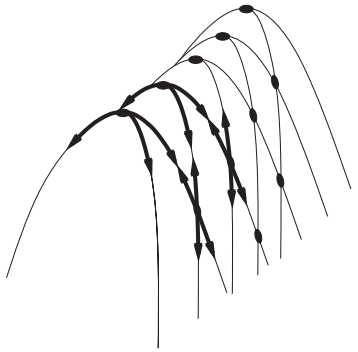
- Properties of HC
  - Like climbing Mount Everest in a fog with amnesia.
  - Hill climbing is incomplete.
  - Time complexity: O(d) where d is the longest path in the space of all solutions.
  - Space complexity: O(b), where b is the branching factor.

How do you know you are done?

# Implementation issues

- Getting stuck in local minima: [randomized restart]
- Design of "neighbors" is critical: ridge problem.
- Value function design is critical: mesa or plateau problem



# More implementation issues

- What to do if there are too many neighbors? Do you need to examine them all to determine the next state? [impatient hill climbers]
- What if there are too few neighbors? Probability of getting stuck in local minima is much higher now. [simulated annealing]
- What if evaluating structures is expensive? Can you use an approximate value function? [caching and interpolation of values]

# Impatient hill climbing

- Select a random neighbor and move there if it is better than the current state, else keep examining random neighbors of current state until you are bored, or you run out of resources.

- function IHC (problem) returns a state
  - current = a random state
  - Do forever
    - next = a random neighbor of current with higher value than current
    - If no neighbor of current is better (examine no more than L neighbors) then return current
    - current = next

# Hill climbing with randomized restart

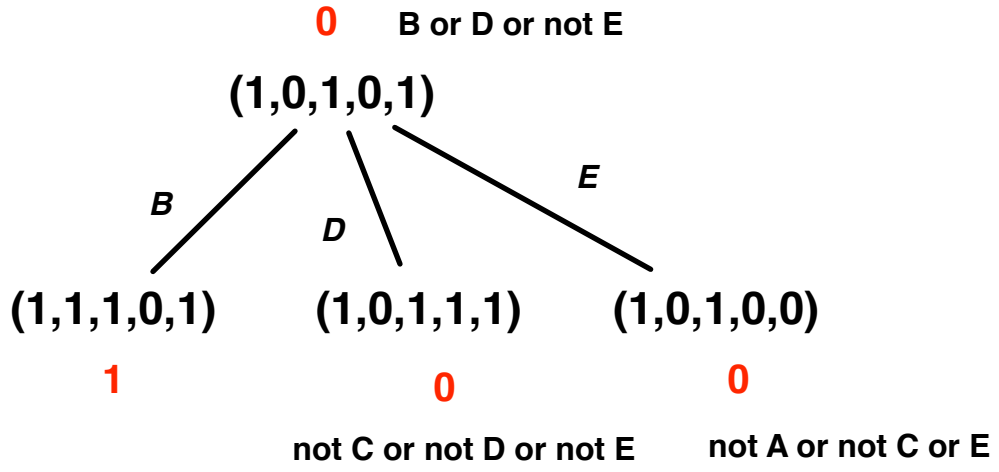
- function HC-RR (problem) returns a state
  - Repeat N times:
    - current = a random state
    - Do forever
      - next = a neighbor of current with the highest value [examine ALL neighbors]
      - If value(next) <= value(current) then return current
      - current = next

- function IHC-RR (problem) returns a state
  - Repeat N times:
    - current = a random state
    - Do forever
      - next = a random neighbor of current with higher value than current
      - If no neighbor of current is better (examine no more than L neighbors) then return current
      - current = next

Randomized hill climbing is incomplete for decision problems, and is not guaranteed to find an optimal solution for optimization problems.

# Neighbor design and value function design

- NASA uses an IHC-RR for solving large scale scheduling problems.
- Need good neighbor design and good value function.

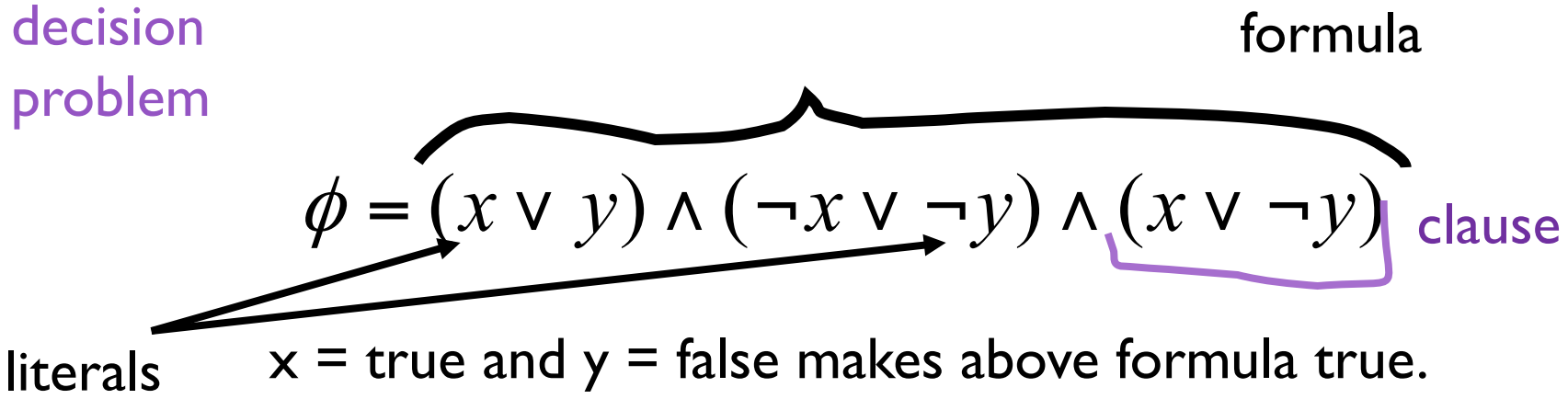



Will this value function do better than the previous one?

# An application of randomized hill climbing to SAT

- The SAT problem: given a formula in the propositional calculus decide if there is an assignment of truth values to its variables that makes the formula true

A decision problem



# Relevance of SAT to AI & CS

- The first problem shown to NP-complete (Cook, 1971).
- Many problems in planning, scheduling, image interpretation, automated reasoning can be formulated as SAT problems.
  - As an example, Schielke (Rice PhD thesis) formulated instruction scheduling in compilers as a SAT problem and explained why simple greedy algorithms suffice for instruction scheduling: something that was unknown to the compiler community!

# The GSAT algorithm

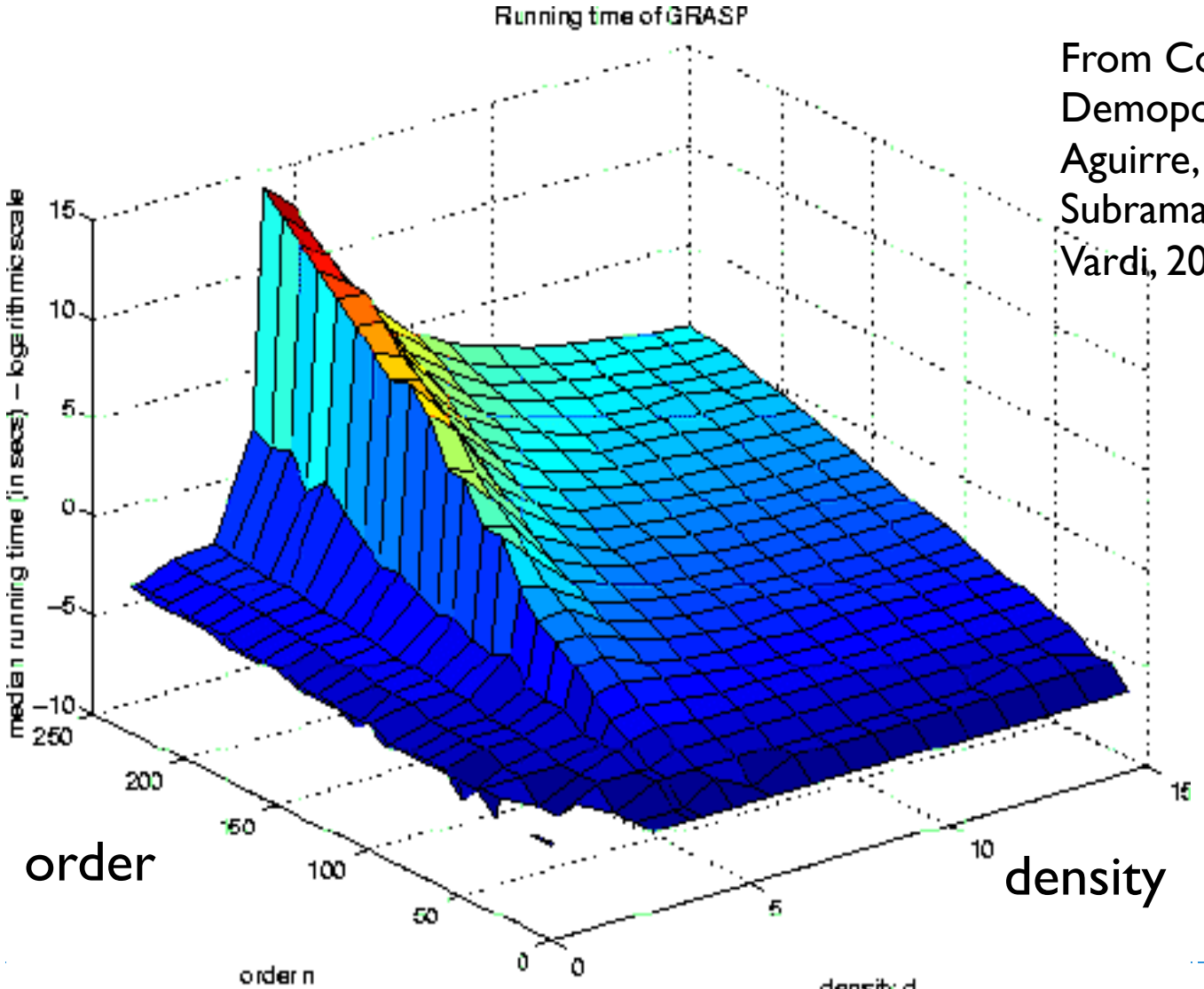
- Function GSAT (formula) returns assignment
  - For i = 1 to num-restarts do
    - current = **random** assignment
    - For j = 1 to L do
      - If current satisfies formula, stop and return it
      - good-Neighbors = all 1 variable flips of current that satisfy more number of clauses in formula than current.
      - If good-Neighbors is empty then consider all 1-variable flips of current that cause the least reduction in number of satisfied clauses in formula, relative to current.
      - Current = **random** element from Good-Neighbors
    - End for
  - End for

Uses two sources of randomization

# The random 3-SAT model

- Two parameter model
  - Number of variables, N (order)
  - Number of clauses, L (length)
- To generate a random 3-SAT formula
  - Select three variables from the N variables.
  - Generate a clause of length 3 from these variables where each variable is negated with a 50% probability
  - Repeat this L times.
- The parameter L/N is called density.

# Difficulty of random 3-SAT problems




From Coarfa, Demopolous, Aguirre, Subramanian Vardi, 2000.

Median running time

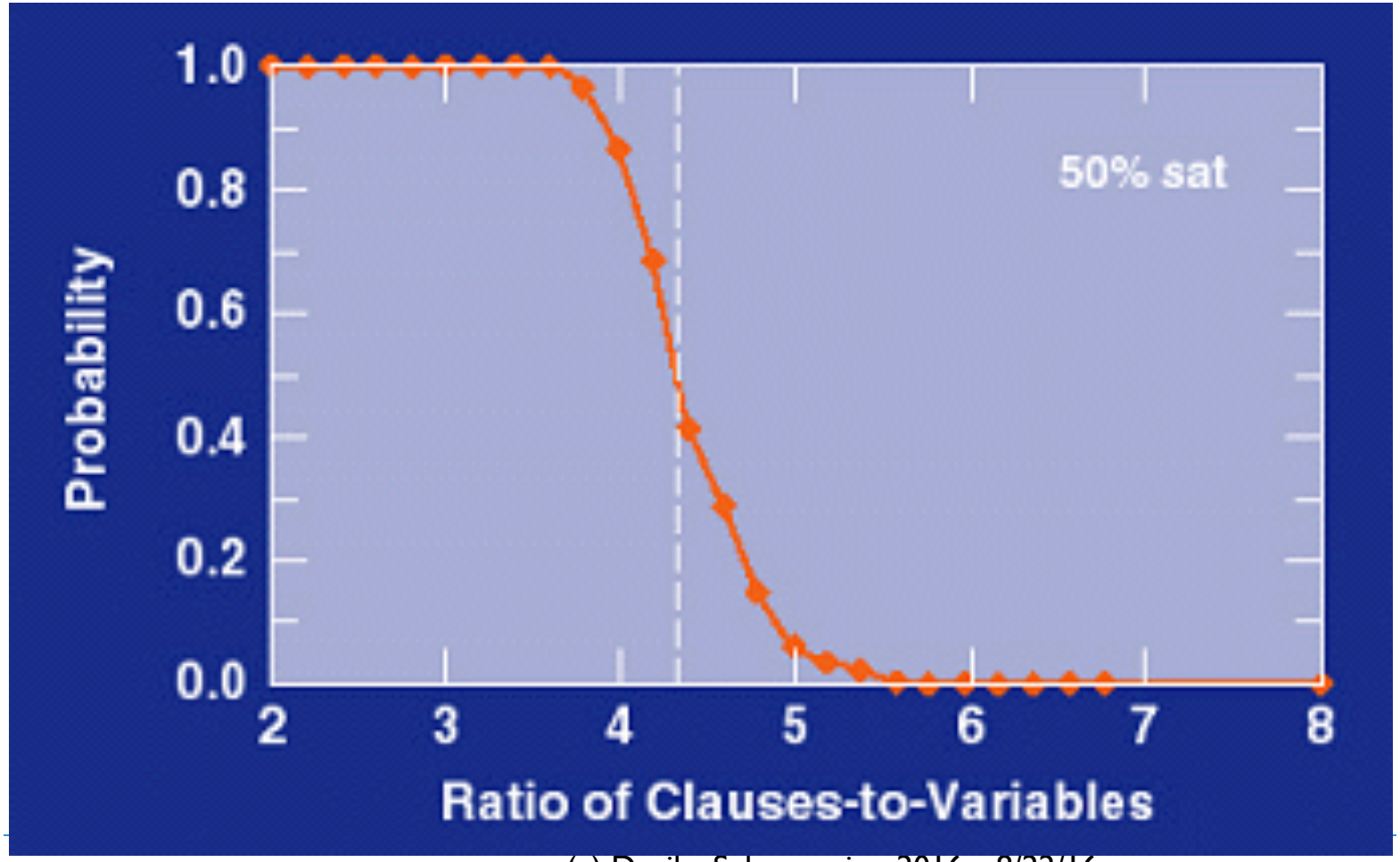
order

density

# Remarks on difficulty of SAT problems

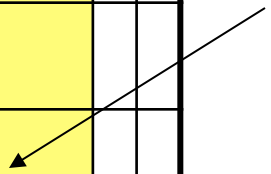
- Very low and very high density problems appear to be "easy". Formulas with few clauses and lots of variables are under-constrained and have many satisfying assignments. Formulas with lots of clauses and few variables are over-constrained and have no satisfying assignments.
- Empirically, the hardest problems appear to occur at density 4.26, and it corresponds to the point where 50% of the random formulas are satisfiable.

# 3 SAT phase transition



# GSAT vs complete algorithm on SAT problems with density 4.26

| Vars | restarts | L | time | DP time | | | |
|---|---|---|---|---|---|---|---|
| 50 | 6.4 | 250 | 0.4s | 11 | 1.4s | | |
| 100 | 42.5 | 500 | 6s | 19 | 2.8m | | |
| 150 | 100.5 | 1500 | 45s | | | | |
| 200 | 248.5 | 2000 | 2.8m | | | | |
| 250 | 268.6 | 2500 | 4.1m | | | | |
| 300 | 231.8 | 6000 | 12m | | | | |
| 400 | 440.9 | 8000 | 34m | | | | |
| 500 | 995.8 | 10000 | 1.6h | | | | |



Selman & Kirkpatrick

# Simulated annealing: intuition

- An approach to handling the problem of being stuck at local maxima.
- A variation on randomized hill climbing in which downhill moves can be made. Makes the search fall off of mediocre local maxima and achieve better local maxima.
- The probability of making a downhill move decreases with time (length of the exploration path from a start state) and decreases with increasing difference in value between current state and neighbor.

# Simulated annealing

- Function simulated-annealing (problem, schedule) returns a solution
  - current = initial state
  - For t = 1 to ∞ do
    - T ← schedule[t]
    - if T = 0 return current
    - next ← a random successor of current.
    - ΔE ← value(next)-value(current)
    - if ΔE > 0 then current = next
    - else current = next with probability $e^{\Delta E/T}$

The probability distribution is derived from the physical process of annealing metals (cooling molten metal to a minimal-energy state). It is called the Boltzmann distribution.

Metropolis, 1953

# Simulated annealing (how it works)

- The probability of making a downhill move is higher when T is higher, and is lower as T comes down. At high temperatures simulated annealing performs a random walk, while at lower temperatures, it is a stochastic hill climber.
- Typically, simulated annealing starts with an initial T, and with each step, T is reduced in a specific way. The recipe for cooling the system is called the annealing schedule.
- Terminate when T gets to zero.
- One of the most widely deployed optimization algorithms in industry.

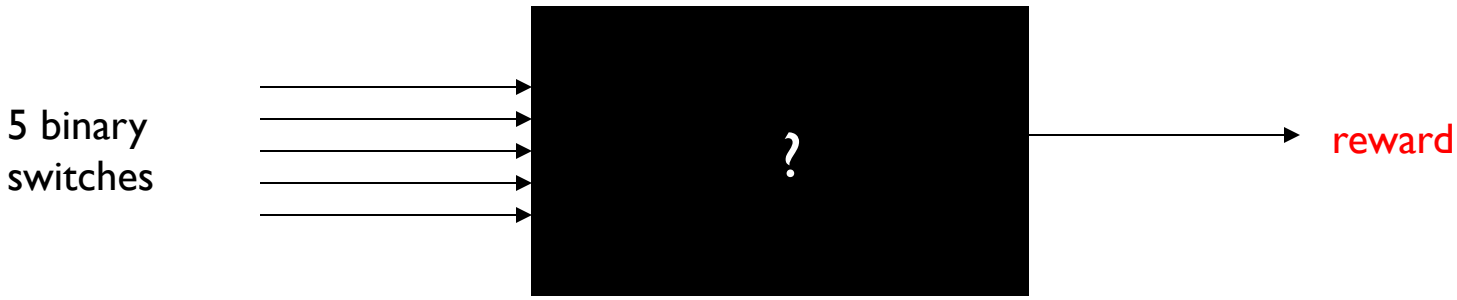
# Design of annealing schedules

- No algorithm for doing this, still an art!
- Empirically, simple annealing schedules tend to work very well. As an example, the annealing schedule that reduces T by a fixed epsilon at each time step yields good results.
- www.ingber.com has a complete study of all known simulated annealing schedules and their performance.

# Genetic algorithms

- Search algorithms based on the mechanics of natural selection.
- A highly simplified computational model of biological evolution.
- Developed by John Holland in the 60s.
- Relies on
  - A population of solutions
    - Survival of the fittest
    - Variations generated by sexual reproduction and mutation.

# A genetic algorithm at work




Find setting of switches that maximizes reward.

# Outline of a GA

- Set up initial population of solutions (generated randomly).
- Generate successive populations using
  - selection
  - crossover
  - mutation
- Repeat generation until no further improvement in reward. The GA is said to have "converged".

# Generating initial population

- Start with a number of random guesses.

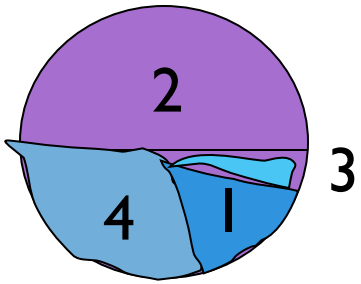
| | | |
|---|---|---|
| 1 | 0 1 1 0 1 | 169 |
| 2 | 1 1 0 0 0 | 576 |
| 3 | 0 1 0 0 0 | 64 |
| 4 | 1 0 0 1 1 | 361 |

- The population size is 4.

**Safety in numbers.**

# Generating successive populations

- Selection: who survives to the next generation?
  - a solution is retained for the next generation in proportion to its reward (fitness).




| | | | |
|---|---|---|---|
| 1 | 0 1 1 0 1 | 169 | 0.14 |
| 2 | 1 1 0 0 0 | 576 | 0.49 |
| 3 | 0 1 0 0 0 | 64 | 0.06 |
| 4 | 1 0 0 1 1 | 361 | 0.31 |

- Analog of "survival of the fittest".

# Selection

- Many methods for doing selection
  - Ranking
  - Tournaments
- All selection methods try to seed the next generation with the "best" individuals from the current one.
- Think of it as a noisy way of selecting the best from a set.
- Selection does not create diversity in a population; it identifies who the most promising mates are.

# Generating successive populations

- The mating pool

| | | |
|---|---|---|
| 1 | 0 1 1 0 1 | 169 |
| 2 | 1 1 0 0 0 | 576 |
| 3 | 1 1 0 0 0 | 576 |
| 4 | 1 0 0 1 1 | 361 |

- 2 copies of the best solution in mating pool, and worst solution is dropped!

# Generating successive populations

- Crossover
  - Pairs of solutions are chosen randomly from mating pool and crossed over at a randomly selected crossover point.
  - Analog of sexual reproduction.

# Crossover

- Many possible crossover operators
  - K-point crossover (generalization of 1 point crossover shown on previous slide). K-point crossover is more disruptive.
  - Uniform crossover: no crossover point. Line up parents and then with some probability p, swap positional values between them.

# Generating successive populations

- Crossover combines elements in two good solutions to generate even better ones.

| | Mating pool | | mate | New population | |
|---|---|---|---|---|---|
| 1 | 0 1 1 \| 0 1 | 169 | 2 | 0 1 1 0 0 | 144 |
| 2 | 1 1 0 \| 0 0 | 576 | 1 | 1 1 0 0 1 | 625 |
| 3 | 1 1 0 \| 0 0 | 576 | 4 | 1 1 0 1 1 | 729 |
| 4 | 1 0 0 \| 1 1 | 361 | 3 | 1 0 0 0 0 | 256 |

Average fitness of new population = 439
Average fitness of initial population = 293

# Generating successive populations

- Mutation
  - each bit in each solution is flipped with a very small probability.
  - Analog of mutation in nature.
  - Insurance policy against premature loss of important subparts of a solution. Performs a random walk in the space of solutions.

# Termination

- When do we stop the GA?
  - After there is little to no improvement in the overall fitness of the population.
  - After a fixed number of iterations.
  - When the population is homogeneous (all individuals in the population are the same).
  - Others?

# Summary of GA parameters

- Population size.
- Selection policy (elitism,…)
- Crossover policy (1-point, uniform,…)
- Mutation rate
- Termination criteria

GA parameter tuning is a huge area of research. Hand tuning of parameters is common in applications. Some rules of thumb exist.

# Rules of thumb for GA parameters

- Population size: 50-100 is typical. Is related to encoding length.
- Selection: fitness proportional selection with 5-20% elitism.
- Crossover: simpler crossover operators preferred; need to ensure that meaningful variations are generated by the process.
- Mutation rate: set very low. If you want 1 bit in 50 to change, set it to 2% (0.02).

# GAs deconstructed…

- Selection: by itself, a somewhat inefficient way of selecting the best from a set (it is stochastic).
- Crossover: by itself, a random shuffle of two solutions.
- Mutation: by itself, a random walk in the space of solutions.
- Note:
  - Selection + mutation = randomized hill climbing
  - Selection + crossover + mutation = GA

# GAs vs randomized hill climbing

- Randomized hill climbing generates variations by mutations and selects good variations randomly.
- When we add crossover to randomized hill climbing (by maintaining a population of solutions), we have the ability to jump to potentially interesting parts of the solution space. Mutation makes sure we can recover key bits damaged by crossover.

# Why do GAs work?

- Independent sampling is provided by large populations that are initialized randomly.
- High fitness solutions are preserved through selection, and this biases the sampling process toward regions of high fitness.
- Crossover combines partial solutions, called "building blocks", thus exploiting the parallelism provided by maintaining a population of solutions.
- Mutation guards against premature loss of diversity in population.

# Embedding HC into a GA

- For a solution
  - 1. Pick a bit in the solution randomly.
  - 2. Flip it, and see if the quality of solution does not decrease. If it doesn't, accept flip, otherwise make no change.
  - 3. Go back to 1 until all positions have been considered.
- If solution is improved by the above process, repeat until no further improvement.

# Some remarks on GAs

- Often the "second best" way to solve a problem. Best used as a way of exploring an unknown solution space.
- Relatively easy to implement.
- Great care is needed to design representation of solutions (e.g., as bit strings), the mutation and crossover operators, as well as the fitness function. In fact, most of the ingenuity in GA design is in these three areas.

# When are GAs inappropriate?

- When exact global optima are needed.
- When any guarantee on quality of solution or convergence time is needed.
- When "appropriate" representations of solutions are not available.

# Extensions to GAs

- Messy GAs: individual solutions represented as variable length strings.
- Genetic programming: individual solutions represented as s-expressions (programs in Scheme or Lisp).
- This field now goes by the name Evolutionary Computation.

# A partial list of GA applications

- Designing jet engines (GE)
- Designing walking strategies for legged robots.
- Scheduling job shop.
- Classifying news stories for Dow Jones.
- Creating art, jazz improvizations.
- TSP.
- Drug design.
- http://creativemachines.cornell.edu/videos