Optimization

Lecture notes for 15-381, 11 April 2000

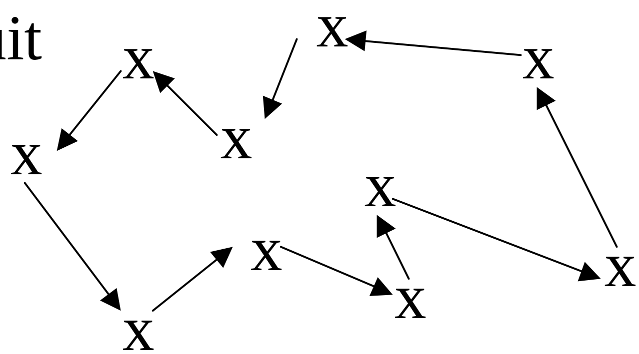
David Cohn <www.cs.cmu.edu/~cohn>

- By the end of this class, you should
  - be able to describe the differences between CSP, ordinary search, and optimization
  - distinguish combinatorial and continuous optimization
  - understand hillclimbing and randomized hillclimbing
  - sort of understand simulated annealing and genetic algorithms
  - describe the amoeba algorithm
  - be able to take a simple problem and code it up for one or more of the above algorithms
  - understand that life is optimization

    (and vice-versa)

Some optimization problems

- traveling salesman - find the shortest circuit



- scheduling - find the shortest schedule
- maximizing number of satisfied clauses

$$\boxed{\begin{array}{l} A \lor \neg B \lor C \\ \neg A \lor C \lor D \\ B \lor D \lor \neg E \\ \neg C \lor \neg D \lor \neg E \\ \neg A \lor \neg C \lor \neg E \end{array}}$$

- find flags that make program run fastest

gcc -o wumpus wumpus.c -f -ffixed -fpack-struct -fregmove -fcommon -ffast-math -fbranch-probabilities-fschedule-insns2 -fschedule-insns -fpretend-float -frerun-loop-opt -frerun-cse-after-loop -fgcse -fdelayed-branch -freg-struct-return -fpcc-struct-return -fcaller-saves -fkeep-static-consts -finline -fkeep-inline-functions -finline-functions -ffunction-cse -fforce-addr -fforce-mem -fpeephole -fwritable-strings -freduce-all-givs -fmove-all-movables -funroll-all-loops -funroll-loops -fstrength-reduce -fthread-jumps -fexpensive-optimizations -fcse-skip-blocks -fcse-follow-jumps -fomit-frame-pointer -fdefer-pop -fvolatile-global -fvolatile -ffloat-store

Optimization as search

- Informally
  - have some structure or parameters being optimized
  - each configuration has some reward/cost
  - want to find max reward/min cost
    - contrast with other search and CSP - they only look for a solution, not an *optimal* solution
  - searching all configurations is intractable
  - **key assumption**: similar configurations have similar costs
- Formally
  - given: state set *S*, move set *moveset(s)*, reward function *eval(s)*
  - find state with maximal reward

Combinatorial vs. continuous optimization

- Continuous
  - clear notion of “neighborhood” and “direction”
  - infinite number of states
- Combinatorial
  - “neighborhood” depends on move set
  - no notion of “direction”
  - finite, but effectively infinite number of states

Discrete optimization

- Iterative improvement search
  - a form of hillclimbing -maximize *eval(s)* by moving to $argmax_{s' \in moveset(s)}\ eval(s')$
    - if all neighboring states are lower, we're stuck, possibly at a local optimum
  - start at a random state, hillclimb until all neighboring states are lower
    - means we're in a (possibly) local optimum
  - select another random state, restart

- Advantages
  - dirt simple to program
  - requires little or no memory
- Disadvantages
  - not terribly clever or robust
  - all cleverness comes fromhaving a good moveset

Selecting a moveset

- Made assumption that similar states have similar values
  - "similar" defined by moveset
  - if moveset is huge, search will be inefficient
  - if moveset is too small, program can get stuck

- What are some movesets for
  - compiler flag optimization?
  - travelling salesman problem?
  - schedule optimization?

Being a little more clever

- Randomized hillclimbing
  - at each step, pick *s'* at random from *moveset(s)*
  - if *eval(s')* > *eval(s)*, move to *s'*
- Some issues
  - how to decide when to stop?
  - any obvious pros and cons with respect to IIS?
  - does this help with local maxima?

Example: GSAT and WALKSAT

- General SATisfiablility
  - given set of variables (A,B,C,D,E) and clauses

    $A \vee \neg B \vee C$
    $\neg A \vee C \vee D$
    $B \vee D \vee \neg E$
    $\neg C \vee \neg D \vee \neg E$
    $\neg A \vee \neg C \vee \neg E$
  - find assignments that satisfy all clauses (or as many as possible)
    - *s = (1,0,1,0,1)*
    - *eval(s) = #satisfied clauses*
    - *moveset(s) = [flip any one variable in s]*
- WALKSAT (randomized GSAT)
  1. pick random unsatisfied clause
  2. consider three moves: flip variable 1,2, or 3
  3. if any improve *eval(s),* accept best
  4. if none improve *eval(s)*, flip a coin:

     50% of time: pick “least bad”
     50% of time: pick one at random
- This is the cutting edge algorithm for GSAT!

Simulated annealing

- Basic idea - be willing to take some downhill steps to get out of local optima
  1. s := initial state
  2. $e := eval(s)$
  3. i:= random move from *moveset(s)*
  4. $e_i := eval(move(s,i))$
  5. if $e_i > e$ then

     $s := move(s,i)$

     $e := e_i$

     else with probability p

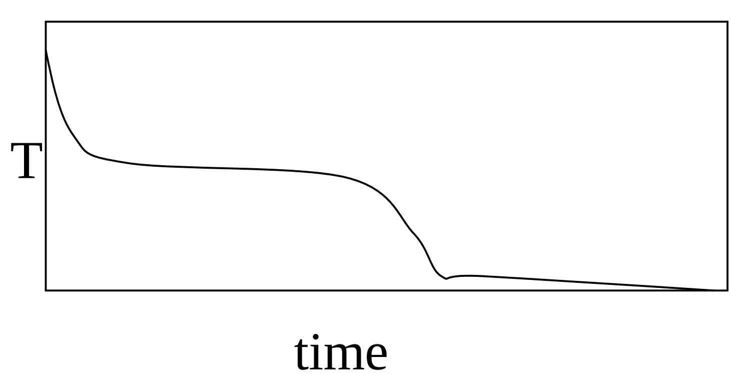
     $s := move(s,i)$

     $e := e_i$
  6. goto (3)

- *Big question: how to set p?*
  - *p = 0.1?*
  - p depends on e-e_i?
  - p depends on time?

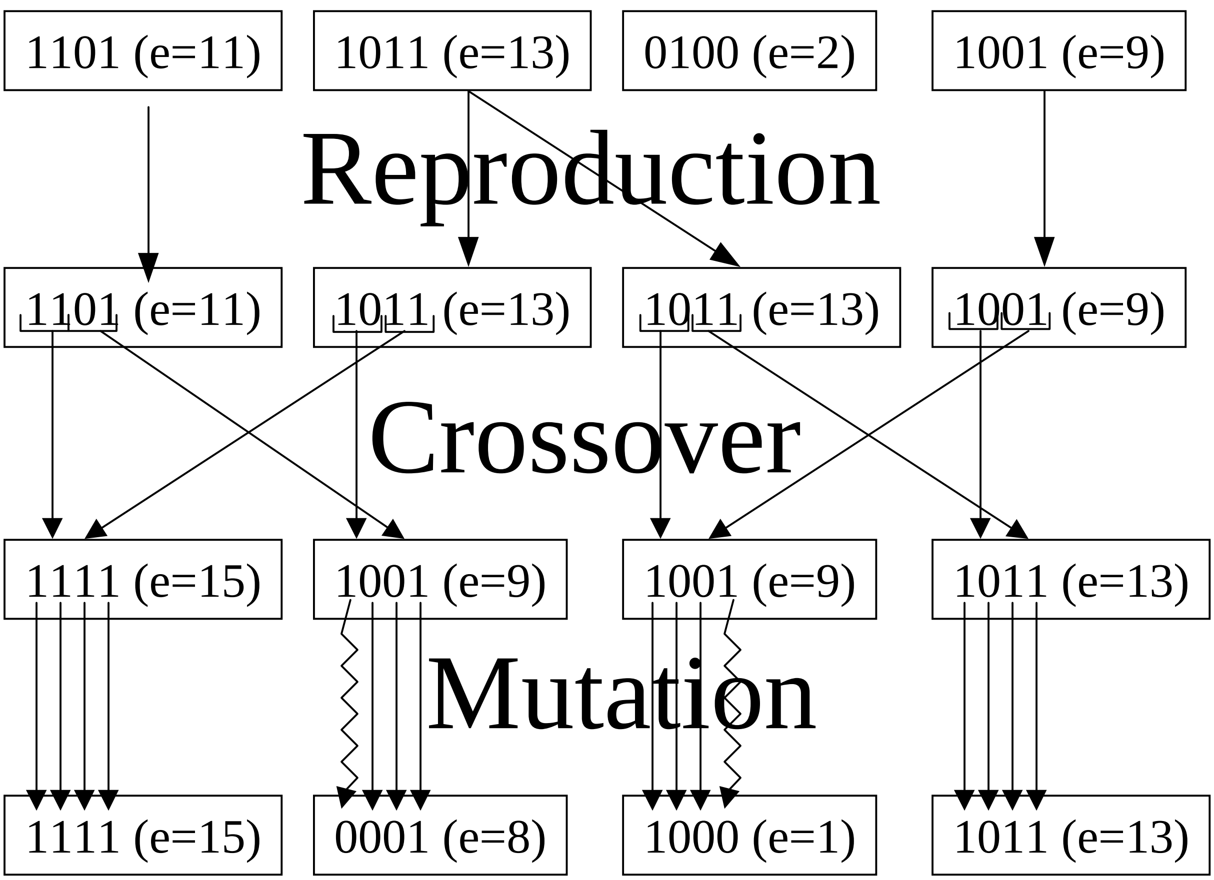
Simulated annealing

- If $e_i >= e$ then definitely accept move
- If $e_i < e$ accept with probability
  $$exp[(e_i - e)/T_i]$$
  - T_i is a "temperature" parameter that gradually decreases with time
  - typically, $T_{i+1} := T_i \cdot r$
- Behavior:
  - high temp - random walk (accept all moves)
  - low temp: stochastic hillclimbing
  - given the "right" cooling schedule, guaranteed to converge to *global* optimum
    - but "right" schedule is insufferably slow; usually use much faster schedule schedule, without guarantees
    - example: "modified Lam" schedule
      - raise and lower temp to meet target accept rate



Genetic algorithms

- Technical idea: “goodness” of solutions may be modular
  – recombine parts of good solutions in hopes of finding better ones
- Illustration (lifted from AWM, lifted from Dean)
  – code up states as bit strings
  – let states “reproduce” and “evolve” to find states with large value of *e*



Genetic algorithm issues

- State representation is critical
  - how to meaningfully encode TSP?
- Crossover/mutation selection is critical
  - how to avoid creating “nonsense” states?
- Representation need not be bitstrings, Could be
  - arbitrary alphanumeric strings
  - terms from a grammar representing functions or programs
    - called “genetic programming”

Combinatorial optimization issues

- "Often, second best way to solve a problem"
  - Moore
- Easy to implement, but sometimes requires deep understanding of solution space
- Most useful on domains that are NP-complete
  - don't use these methods if the problem can be solved using linear programming, A*, or constraint propagation
- Very wasteful in terms of evaluations - may evaluate same state over and over again
  - some techniques like "tabu search" keep list of recent already-evaluated states to avoid re-evaluating

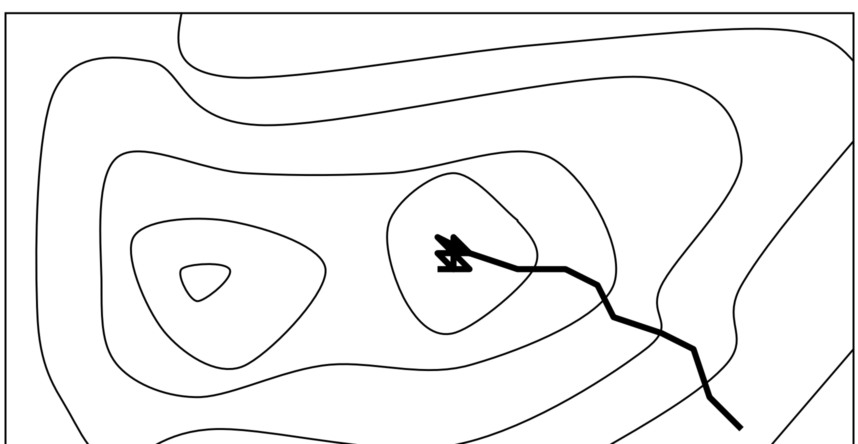
Continuous optimization

- State space is continuous
  - have notion of “neighborhood”, “direction”
  - moveset is more intuitive

- Examples
  - setting baking time, temp, and cocoa content to maximize chocolate quality
  - setting power, pitch, altitude to maximize airspeed or fuel efficiency
  - setting weights in a neural network to maximize accuracy on a training set

Continuous optimization techniques

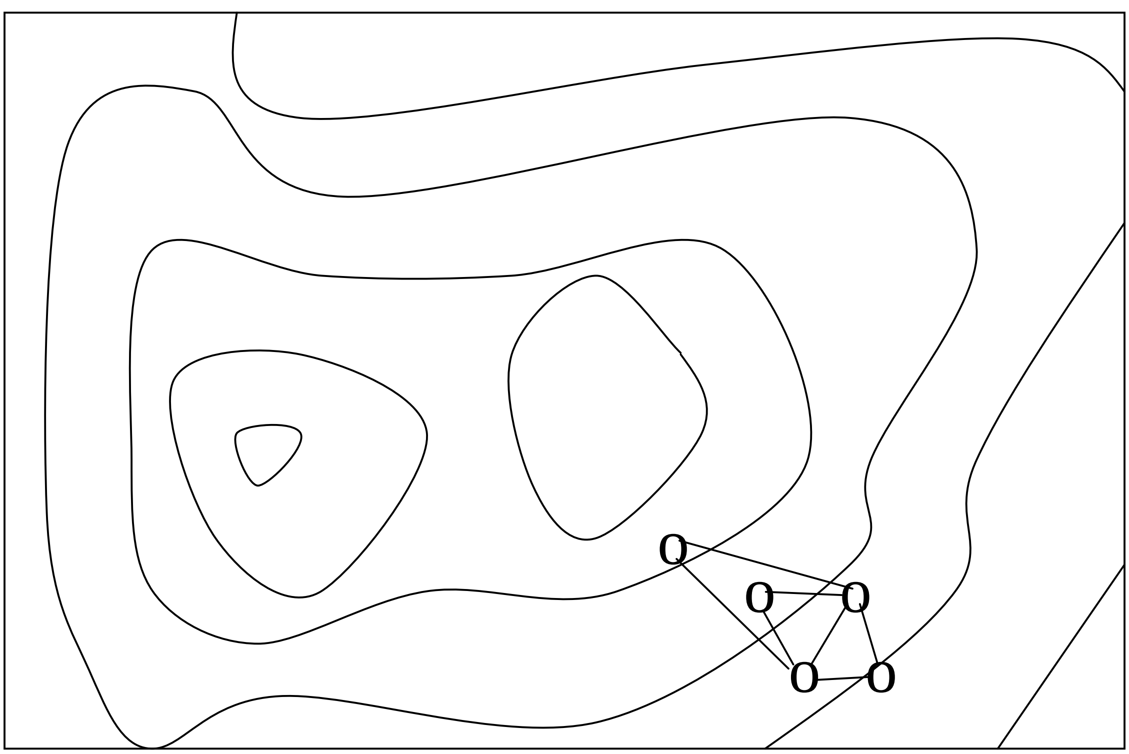
- Most combinatorial optimization techniques can be adapted to continuous domains
- Some additional techniques become applicable
  - gradient descent
  - amoeba
  - pmax
  - Q2
  - many others...



- If we can compute a gradient, can hillclimb to local optimum

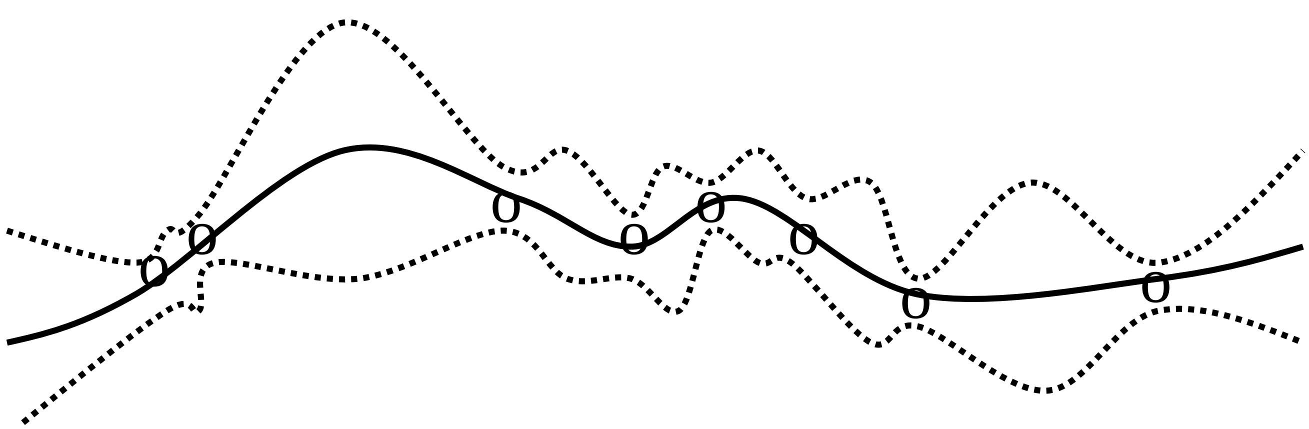
Amoeba optimization

- A.k.a "downhill simplex method"
  - in n-dimensional space, evaluate n+1 similar points - these form a "simplex"
  - project the worst point through the "face" formed by the n other points
  - if the new point is better, try extending projection
  - if new point is worse, try interpolating
  - if all else fails, shrink points toward mean, retry



PMAX

- Use evaluated states as input to function fitter, such as neural network, or nonlinear regression
  – train fitter on evaluated points
  – hillclimb on model, using gradient descent
  – can compute confidence intervals from model
- Which point to evaluate next?
  – estimated best? - might hallucinate
  – optimistic best? - might be too eager to explore everything
  – pessimistic best? - might be too conservative



Issues in optimization

- Noise, noise, noise
  - some algorithms can be made robust to noise, some just fall apart
  - which are which? hillclimbing, simulated annealing, GA, gradient descent, amoeba, PMAX?
  - how would you “robustify” an algorithm?
- Dimensionality
- Experiment costs vs. deciding where to evaluate next
  - if evaluations are expensive, want to make every one count
  - if evaluations are cheap, willing to waste a few to not have to think too hard