Local Search

The 2-SAT Problem

Algorithms: Design and Analysis, Part II

2-SAT

Inputs: ① n Boolean variables $x_1, x_2, \ldots, x_n$ (can be set to TRUE or FALSE)

② m clauses of 2 literals each ("literal" = x_i or $\neg x_i$)

Example: $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_4)$

Output: "yes" if there is an assignment that simultaneously satisfies every clause, "no" otherwise.

Example: "yes", via (e.g.) $x_1 = x_3 = \text{TRUE}$

and $x_2 = x_4 = \text{FALSE}$

(In)Tractability of SAT

2-SAT: Can be solved in polynomial time!

- reduction to computing strongly connected components
- "backtracking" works in polynomial time
- randomized local search (next)

non-trivial exercises

3-SAT: Canonical NP-complete.

- brute-force search $\approx 2^n$ time
- Can get time $\approx \left(\frac{4}{3}\right)^n$ via randomized local search [Schöning '02]

Papadimitriou's 2-SAT Algorithm

Repeat $\log_2 n$ times:
- Choose random initial assignment
- repeat $2n^2$ times:
  - if current assignment satisfies all clauses, halt + report this
  - else, pick arbitrary unsatisfied clause and flip the value of one of its variables [choose between the two uniformly at random]

Report "unsatisfiable".

n = number of variables

key question: if there's a satisfying assignment, will the algorithm find one (with probability close to 1)?

Obvious good points

1. runs in polynomial time
2. always correct on unsatisfiable instances.

Tim Roughgarden