Artificial Intelligence

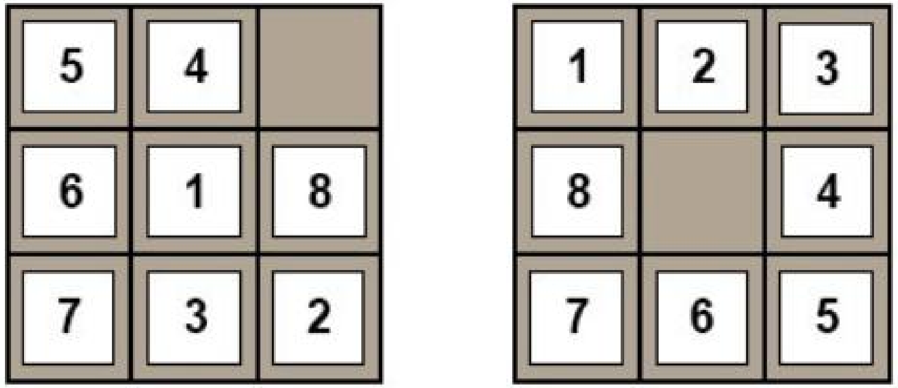
CS4365 --- Fall 2022

Local Search

Instructor: Yunhui Guo

Search Algorithms

- BFS, DFS, UCS, A* etc
    - Explore the state-space systemantically to the goal state

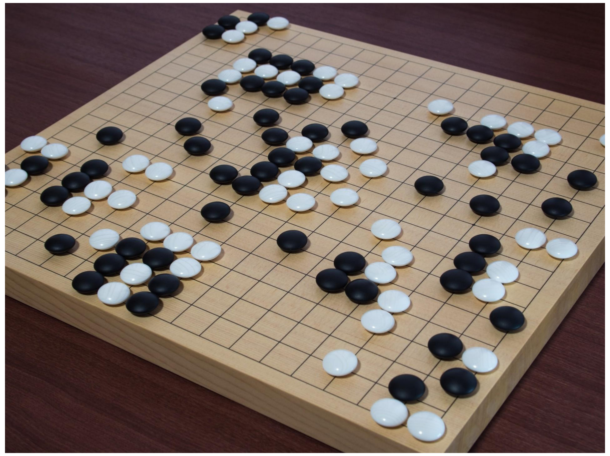


Start State Goal State

- Need to reconstruct the path

Local Search Methods

- The search spaces for some real-world problems is enormously big
  - How many combinations can 500 processes be assigned to 100 computers? 100^{500} states
  - There are 3^{361} states in Go board



- A completely different kind of method is called for:
  - Local Search Methods

Local Search Methods

- Applicable when we're interested in the Goal State -- not in how to get there
  - E.g. N-Queens, Course Scheduling Problem or Job-shop Scheduling

- Basic idea:
  - use a single current state
  - don't save paths followed
  - generally move only to sucessors/neighbors of that state
  - find the best state according to an objective function or cost function

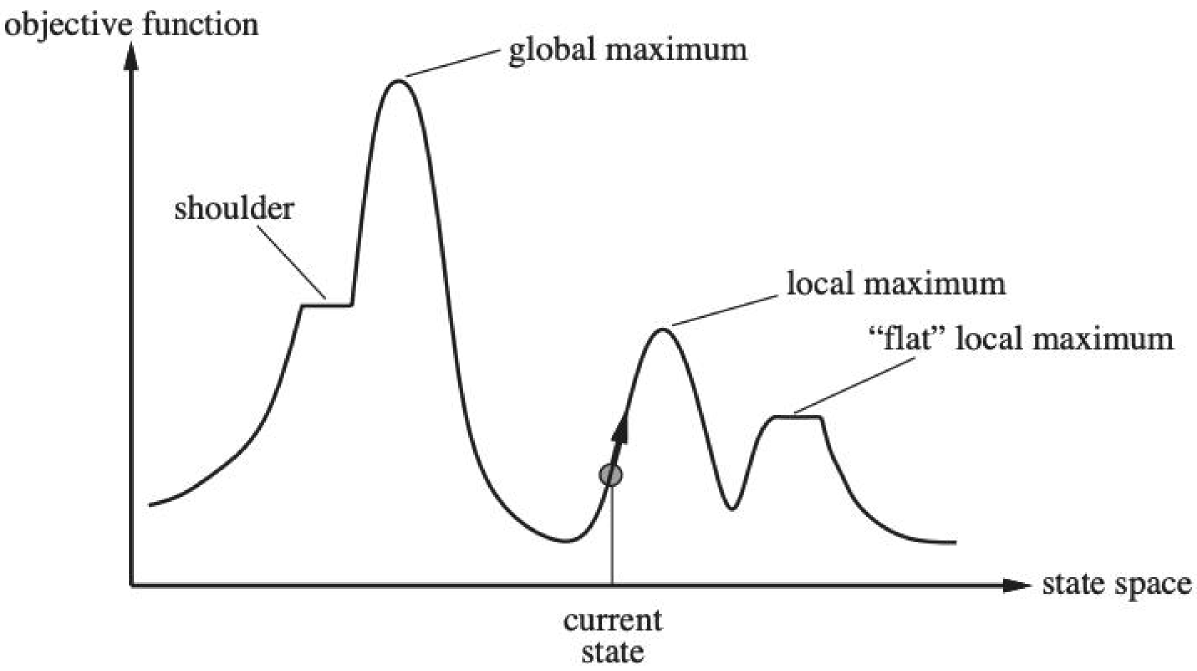
Local Search Methods

- Advantages:
  - Use very little memory
  - Can be applied to problems with a large number of states
  - Can be applied to problems with changing state spaces

- Disadvantages:
  - Cannot recover the path
  - The solution may not be optimal

State-Space Landscape

- Local Search Methods rely on an objective function or a cost function to compute the state-space landscape



- Local search algorithms explore this landscape

Hill Climbing Search

- Move in the direction of increasing value
- Terminate when it reaches a “peak” where no neighbor has a higher value.

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of *current*
    if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
    current ← *neighbor*

Hill Climbing Search

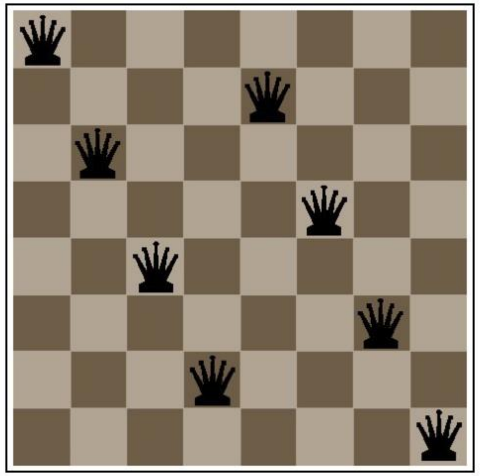
Hill Climbing Search

- Only need to record the state and the value of the objective function or cost function

- Similar to greedy algorithm, only check the value of the immediate neighbors

- Rely on complete-state formulation instead of incremental formulation

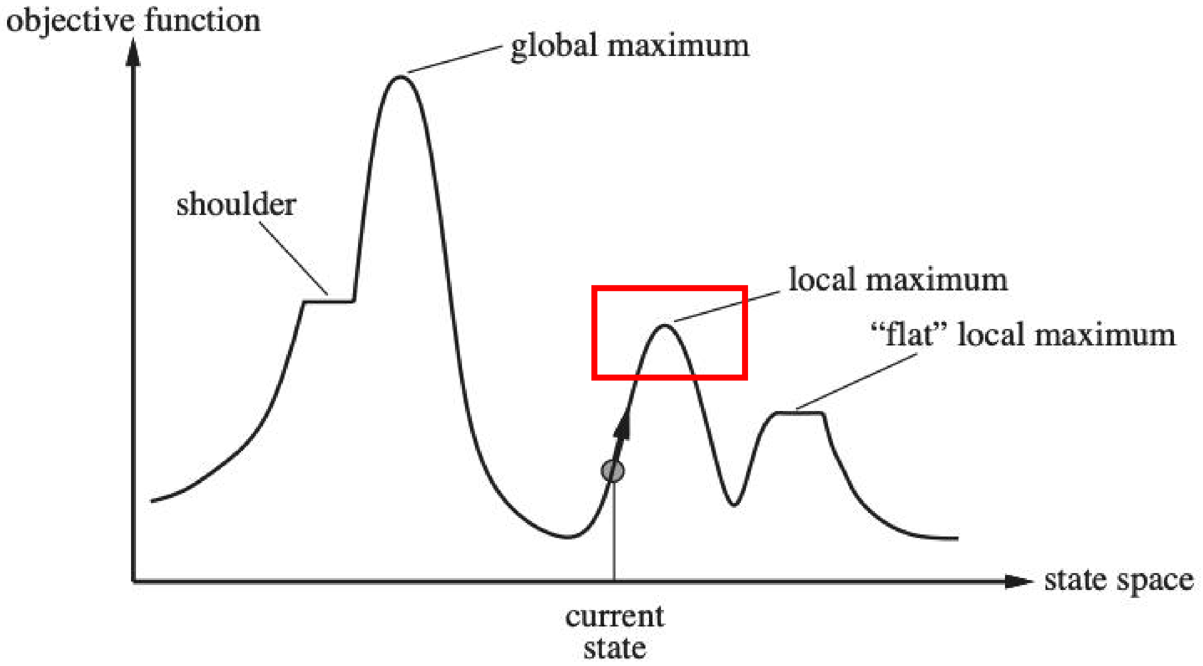
Complete-State Formulation vs. Incremental Formulation



- Complete-State Formulation
  - All queens are in the board. Move any queen in the same column
  - Why used in local search?
- Incremental Formulation
  - Add each queen to the board one by one (used in many search algorithms we covered before)

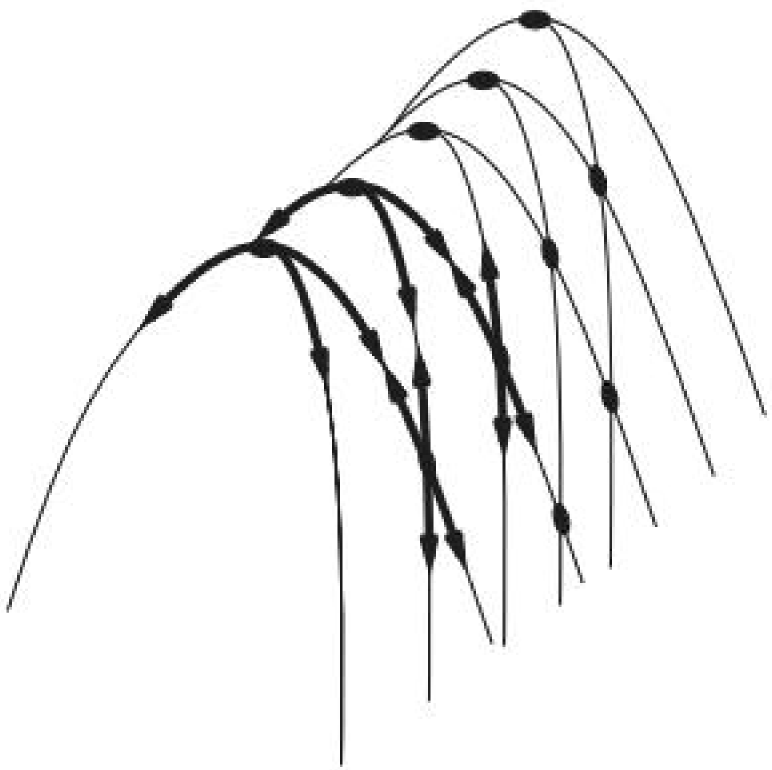
Hill Climbing Search Can Get Stuck

- Local maxima
  - A peak that is higher than all its neighbors but lower than the global maximum.



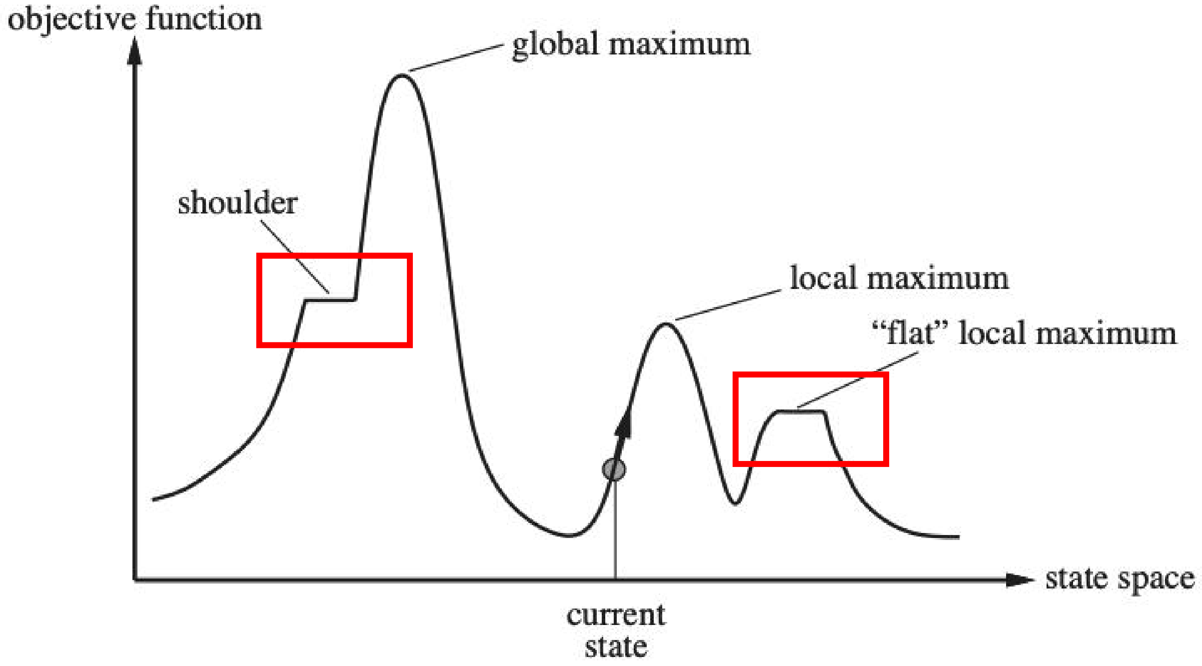
Hill Climbing Search Can Get Stuck

- Ridges
  - A sequence of local maxima



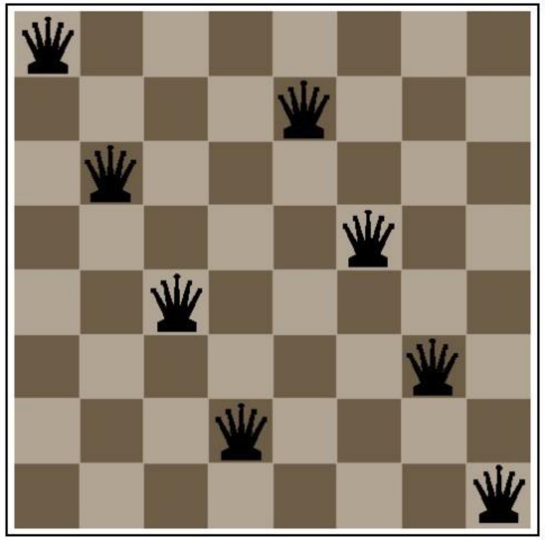
Hill Climbing Search Can Get Stuck

- Plateaux or shoudlers
  - The objective function is constant



Example: 8-queen problem

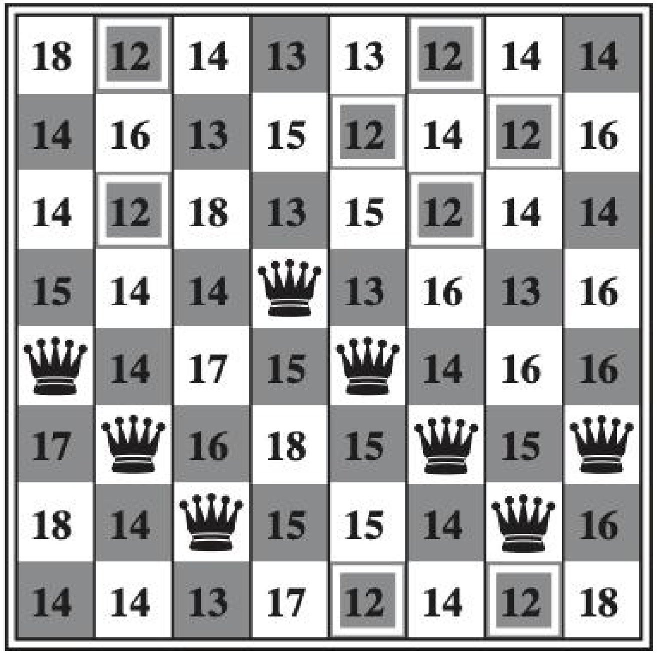
- Put n queens on an n × n board with no two queens on the same row, column, or diagonal



- What is the state-space of this problem?
- How is this problem different from the 8-puzzle problem?

Example: 8-queen problem

- Heuristic cost function: the number of pairs of queens that are attacking each other, either directly or indirectly

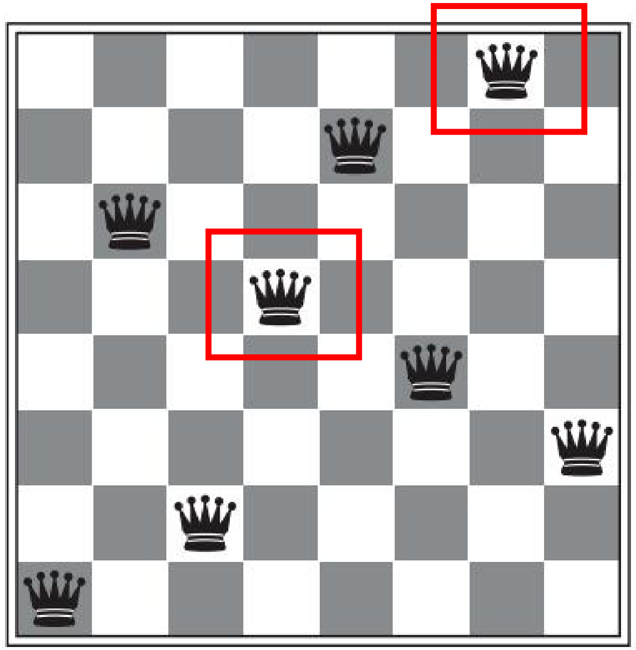


$h = 17$

- Each time we can move the queen to another square in the same column

Example: 8-queen problem

- Heuristic cost function: the number of pairs of queens that are attacking each other, either directly or indirectly



- Local minumum with h =1

Example: Satisfiability

- Propositional logic:
  - Literals (True or False)

    A, B, C, ...
  - Logical connectives
    - And: $\wedge$ Or: $\vee$ Not: $\neg$

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$
false	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*
true	*false*	*false*	*false*	*true*
true	*true*	*false*	*true*	*true*

  - Clause: a disjunction of literals

    $$l_1 \vee \ldots \vee l_n$$
  - Conjunctive normal form:

    $$(A \vee B \vee C) \wedge (\neg B \vee C \vee D) \wedge (A \vee \neg C \vee D)$$

Example: Satisfiability

- A wide variety of key CS problems can be translated into a propositional logical formalization

  e.g., $(A \vee B \vee C) \wedge (\neg B \vee C \vee D) \wedge (A \vee \neg C \vee D)$

- Solved by finding a truth assignment to the propositional variables $(A, B, C, ...)$ that makes it true, i.e., a model.

- If a formula has a model, we say that it is “satisfiable”

Example: Satisfiability Testing

- Best-known method: Davis-Putnam Procedure (1960)

  - Backtrack search (DFS) through the space of truth assignments

- Assigning values to variables one by one and simplifying at each step

- Cannot be satisfied

$$A \wedge \neg A$$

Example: Satisfiability Testing

Best-known method: Davis-Putnam Procedure (1960)

$$(A \vee C) \wedge (\neg A \vee C) \wedge (B \vee \neg C) \wedge (A \vee \neg B)$$

Example: Satisfiability Testing

Best-known method: Davis-Putnam Procedure (1960)

$$(A \vee C) \wedge (\neg A \vee C) \wedge (B \vee \neg C) \wedge (A \vee \neg B)$$

Example: Satisfiability Testing

Best-known method: Davis-Putnam Procedure (1960)

$$(A \vee C) \wedge (\neg A \vee C) \wedge (B \vee \neg C) \wedge (A \vee \neg B)$$

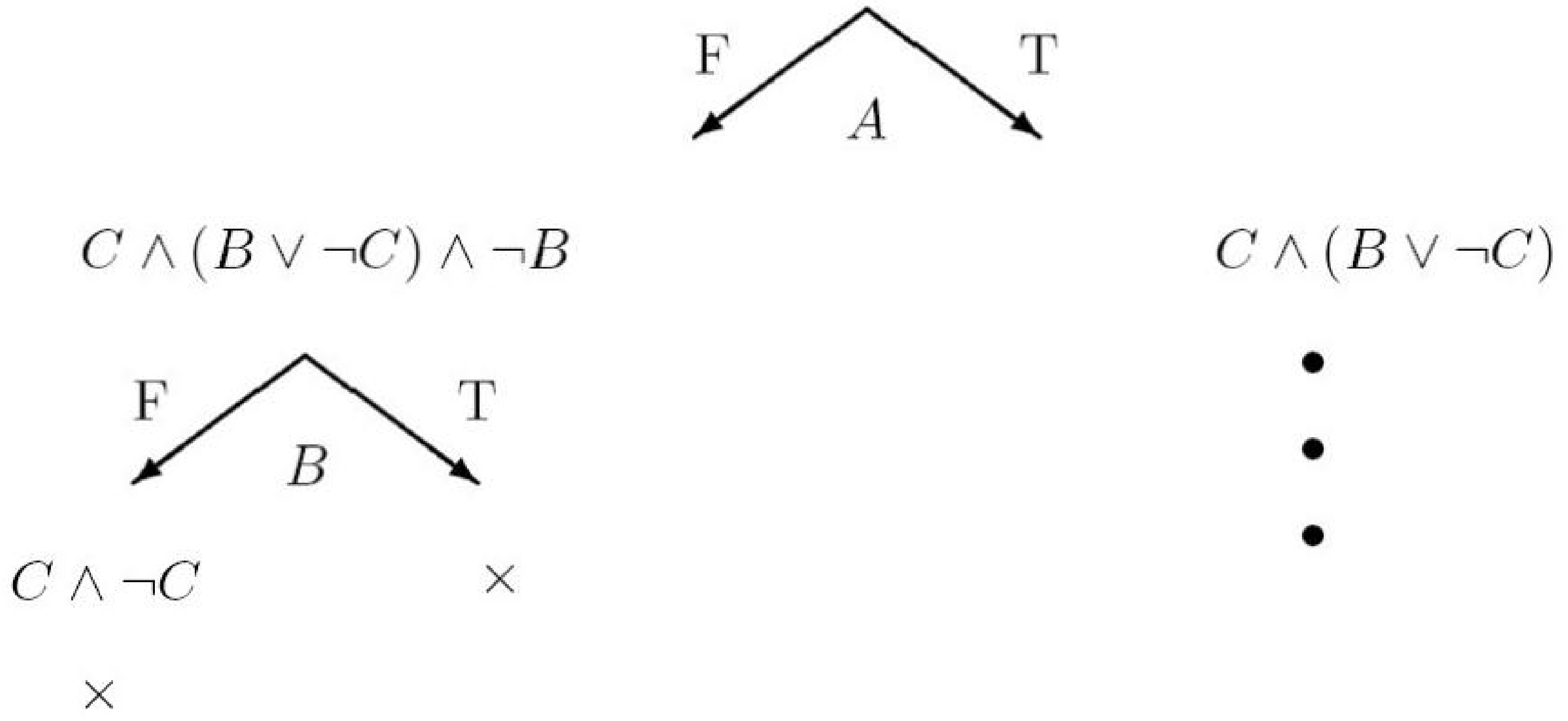
$$C \wedge (B \vee \neg C) \wedge \neg B$$

$$C \wedge (B \vee \neg C)$$

Example: Satisfiability Testing

Best-known method: Davis-Putnam Procedure (1960)

$$(A \vee C) \wedge (\neg A \vee C) \wedge (B \vee \neg C) \wedge (A \vee \neg B)$$



Example: Satisfiability Testing

- To date, Davis-Putnam Procedure is still the fastest sound and complete method.
- However, there are classes of formulas where the procudure scales badly.
- Consider an imcomplete local search procudure

Greedy Local Search -- GSAT

- Begin with a random truth assignment (assume CNF).
- Flip the value assignment to the variable that yields the greatest number of satisfied clauses. (Note: Flip even if there is no improvement.)
- Repeat until a model is found, or have performed a speficied maximum number of flips.
- If a model is still not found, repeat the entire process, starting from a different initial random assignment.

Greedy Local Search -- GSAT

- Input: a conjunctive normal form α, MAX-FLIPS, MAX-TRIES
- Output: a satisfying truth assignment if found

Greedy Local Search -- GSAT

- Input: a conjunctive normal form α, MAX-FLIPS, MAX-TRIES
- Output: a satisfying truth assignment if found

- For i = 1 to MAX-TRIES:

    T := a randomly generated truth assignment

    For j = 1 to MAX-FLIPS:

        If T satisfies α, then return T

        p := a propositional variable such that a change of its truth leads to the largest number of satisfied clauses

        T := T with the truth value of p reversed

return "no satisfying assignment found"

Greedy Local Search -- GSAT

$$(A \vee C) \wedge (\neg A \vee C) \wedge (B \vee \neg C)$$

- A: False   B: False   C: True

Greedy Local Search -- GSAT

$$(A \vee C) \wedge (\neg A \vee C) \wedge (B \vee \neg C)$$

- A: False   B: False   C: True

A	B	C	$(A \vee C)$	$\wedge$	$(\neg A \vee C)$	$\wedge$	$(B \vee \neg C)$	Score
F	F	F	$\times$		$\surd$		$\surd$	2
F	F	T	$\surd$		$\surd$		$\times$	2
F	T	T	$\surd$		$\surd$		$\surd$	3

How well does it work?

- First intuition: It wil get stuck in local minima, with a few unsatisfied clauses.

- Note we are not interested in almost satisfying assigments
  - E.g., a plan with one “magic” step is useless.
  - Contrast with optimization problems.

- GSAT is not complete.

- Surprise: It often finds global minimum!
  - i.e., finds satisfying assigments

How well does it work?

- Generate a large number of random formulas
  - Different number of propositional variables
  - Different number of clauses

$$(A \lor C) \land (\neg A \lor C) \land (B \lor \neg C)$$

- Run GSAT and Davis-Putnam Procedure on the generated formulas and measure the time to find the right assignment.

Greedy Local Search -- GSAT

formulas		GSAT			DP		
vars	clauses	M-FLIPS	tries	time	choices	depth	time
50	215	250	6.4	0.4s	77	11	1.4s
70	301	350	11.4	0.9s	42	15	15s
100	430	500	42.5	6s	84×10^3	19	2.8m
120	516	600	81.6	14s	0.5×10^6	22	$\boxed{18m}$
140	602	700	52.6	14s	2.2×10^6	27	$\boxed{4.7h}$
150	645	1500	100.5	45s	—	—	—
200	860	2000	248.5	2.8m	—	—	—
250	1062	2500	268.6	4.1m	—	—	—
300	1275	6000	231.8	12m	—	—	—
400	1700	8000	440.9	34m	—	—	—
500	2150	10000	995.8	1.6h	—	—	—

Limitations of GSAT

$$(A \vee \neg B \vee C) \wedge (A \vee \neg C \vee D) \wedge (A \vee \neg D \vee \neg B) \wedge (A \vee E \vee B) \wedge (A \vee \neg E \vee B)$$
$$... \wedge (\neg A \vee \neg X \vee Y) \wedge (\neg A \vee \neg Z \vee P)$$

- Can be satisfied only when A is True
- But GSAT prefers a negative assignment

Limitations of GSAT

$$(A \vee \neg B \vee C) \wedge (A \vee \neg C \vee D) \wedge (A \vee \neg D \vee \neg B) \wedge (A \vee E \vee B) \wedge (A \vee \neg E \vee B)$$

- When A is False

$$\Downarrow$$

$$(\neg B \vee C) \wedge (\neg C \vee D) \wedge (\neg D \vee \neg B) \wedge (E \vee B) \wedge (\neg E \vee B)$$

- When B is True

$\Rightarrow$ $C \wedge (\neg C \vee D) \wedge \neg D$ Unsatisfied

- When B is False

$\Rightarrow$ $(\neg C \vee D) \wedge E \wedge \neg E$ Unsatisfied