Local Search

CS311
David Kauchak
Spring 2013

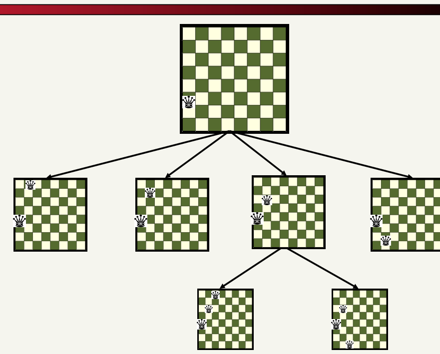
Some material borrowed from:
Sara Owsley Sood and others

Administrative

- Assignment 2 due Tuesday before class
- Written problems 2 posted
- Class participation
- http://www.youtube.com/watch?v=irHFVdphfZQ&list=UUCDOQrpqLqKVcTCKzqarxLg

N-Queens problem

N-Queens problem



N-Queens problem

What is the depth?

- 8

What is the branching factor?

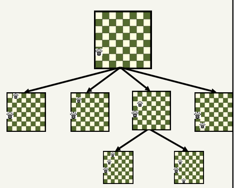
- ≤ 8

How many nodes?

- 8^8 = 17 million nodes

Do we care about the path?

What do we really care about?



Local search

So far a systematic exploration:

- Explore full search space (possibly) using principled pruning (A*, . . .)

Best such algorithms (IDA*) can handle

- 10^{100} states ≈ 500 binary-valued variables (ballpark figures only!)

but. . . some real-world problem have 10,000 to 100,000 variables $10^{30,000}$ states

We need a completely different approach

Local search

Key difference: we don't care about the path to the solution, only the solution itself!

Other similar problems?

- sudoku
- crossword puzzles
- VLSI design
- job scheduling
- Airline fleet scheduling
  - http://www.innovativescheduling.com/company/Publications/Papers.aspx
- …

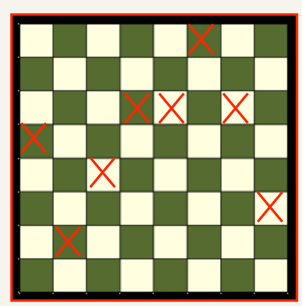
Alternate Approach

Start with a random configuration

repeat

- generate a set of "local" next states
- move to one of these next states

How is this different?



Local search

Start with a random configuration

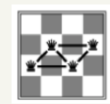
repeat

- generate a set of "local" next states
- move to one of these next states

Requirements:

- ability to generate an initial, random guess
- generate the set of next states that are "local"
- criterion for evaluating what state to pick!

Example: 4 Queens



State:

- 4 queens in 4 columns

Generating random state:

- any configuration
- any configuration without row conflicts?

Operations:

- move queen in column

Goal test:

- no attacks

Evaluation:

- h(*state*) = number of attacks

Local search

Start with a random configuration

repeat

- generate a set of "local" next states
- move to one of these next states

Starting state and next states are generally constrained/specified by the problem

Local search

Start with a random configuration

repeat

- generate a set of "local" next states
- move to one of these next states

How should we pick the next state to go to?

Greedy: Hill-climbing search

Start with a random configuration

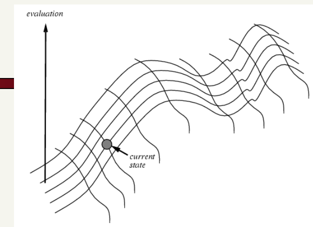
repeat

- generate a set of “local” next states
- move to one of these next states

pick the best one according to our heuristic

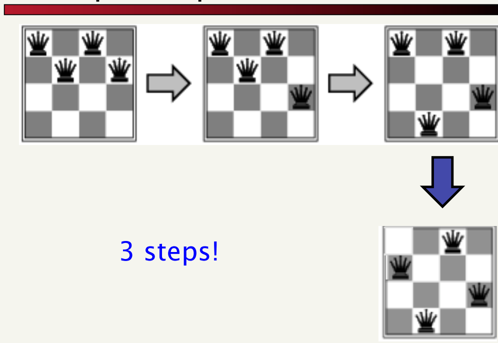
again, unlike A* and others, we don’t care about the path

Hill-Climbing




```
def hillClimbing(problem):
   """ This function takes a problem specification and returns
       a solution state which it finds via hill climbing """
   currentNode = makeNode(initialState(problem))
   while True:
      nextNode = getHighestSuccessor(currentNode,problem)
      if value(nextNode) <= value(currentNode):
         return currentNode
      currentNode = nextNode
```

Example: *n*-queens



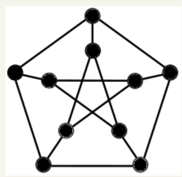
3 steps!

Graph coloring

What is the graph coloring problem?

Graph coloring

Given a graph, label the nodes of the graph with *n* colors such that no two nodes connected by an edge have the same color



Is this a hard problem?

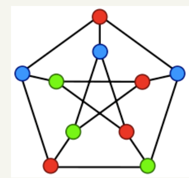
- NP-hard (NP-complete problem)

Applications

- scheduling
- sudoku

Graph coloring

Given a graph, label the nodes of the graph with *n* colors such that no two nodes connected by an edge have the same color



Is this a hard problem?

- NP-hard (NP-complete problem)

Applications

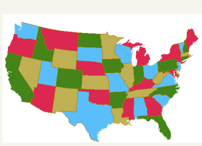
- scheduling
- sudoku

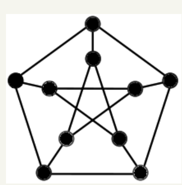
Local search: graph 3-coloring

Initial state?

Next states?

Heuristic/evaluation measure?



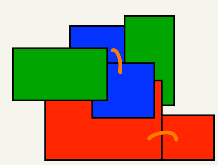


Example: Graph Coloring

1. Start with random coloring of nodes
2. Change color of one node to reduce # of conflicts
3. Repeat 2

Eval: number of “conflicts”, pairs adjacent nodes with the same color:

2

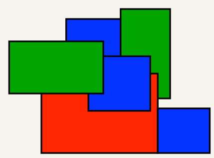


Example: Graph Coloring

1. Start with random coloring of nodes
2. Change color of one node to reduce # of conflicts
3. Repeat 2

Eval: number of “conflicts”, pairs adjacent nodes with the same color:

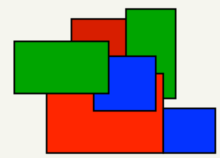
1



Example: Graph Coloring

1. Start with random coloring of nodes
2. Change color of one node to reduce # of conflicts
3. Repeat 2

Eval: number of “conflicts”, pairs adjacent nodes with the same color:

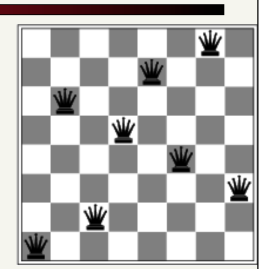


Hill-climbing Search: 8-queens problem

h = number of pairs of queens that are attacking each other, either directly or indirectly

h = 17 for the above state

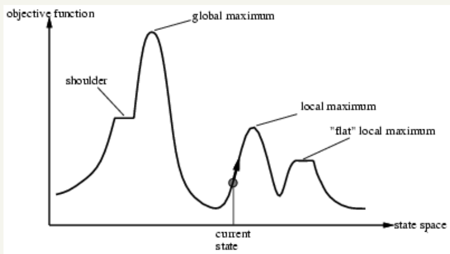
Hill-climbing search: 8-queens problem



86% of the time, this happens

After 5 moves, we’re here… now what?

Problems with hill-climbing



Hill-climbing Performance

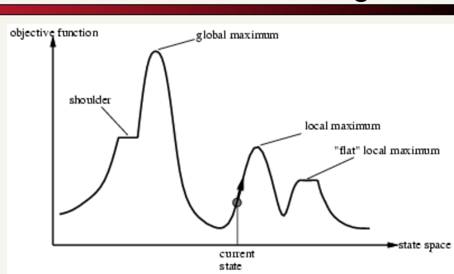
Complete?

Optimal?

Time Complexity

Space Complexity

Problems with hill-climbing



Ideas?

Idea 1: restart!

Random-restart hill climbing

- if we find a local minima/maxima start over again at a new random location

Pros:

Cons:

Idea 1: restart!

Random-restart hill climbing

- if we find a local minima/maxima start over again at a new random location

Pros:

- simple
- no memory increase
- for n-queens, usually a few restarts gets us there
  - the 3 million queens problem can be solve in < 1 min!

Cons:

- if space has a lot of local minima, will have to restart a lot
- loses any information we learned in the first search
- sometimes we may not know we're in a local minima/maxima

Idea 2: introduce randomness

```
def hillClimbing(problem):
    """ This function takes a problem specification and returns
        a solution state which it finds via hill climbing """
    currentNode = makeNode(initialState(problem))
    while True:
        nextNode = getHighestSuccessor(currentNode,problem)
        if value(nextNode) <= value(currentNode):
            return currentNode
        currentNode = nextNode
```

Rather than always selecting the best, pick a random move with some probability

- sometimes pick best, sometimes random (epsilon greedy)
- make better states more likely, worse states less likely
- book just gives one… many ways of introducing randomness!

Idea 3: simulated annealing

What the does the term annealing mean?

"When I proposed to my wife I was annealing down on one knee"?

Idea 3: simulated annealing

What the does the term annealing mean?

Annealing, in metallurgy and materials science, is a heat treatment wherein a material is altered, causing changes in its properties such as strength and hardness. It is a process that produces conditions by heating to above the recrystallization temperature and maintaining a suitable temperature, and then cooling. Annealing is used to induce ductility, soften material, relieve internal stresses, refine the structure by making it homogeneous, and improve cold working properties.

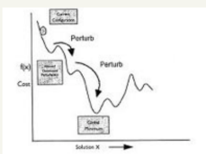
Simulated annealing

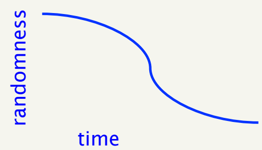
Early on, lots of randomness

- avoids getting stuck in local minima
- avoids getting lost on a plateau

As time progresses, allow less and less randomness

- Specify a “cooling” schedule, which is how much randomness is included over time

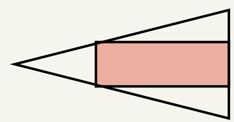




Idea 4: why just 1 initial state?

Local beam search: keep track of *k* states

- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state
  - stop
- else
  - select the *k* best successors from the *complete list and repeat*



Local beam search

Pros/cons?

- uses/utilized more memory
- over time, set of states can become very similar

How is this different than just randomly restarting *k* times?

What do you think regular beam search is?

An aside… Traditional beam search

A number of variants:

- BFS except only keep the top k at each level
- best-first search (e.g. greedy search or A*) but only keep the top k in the priority queue

Complete?

Used in many domains

- e.g. machine translation
  - http://www.isi.edu/licensed-sw/pharaoh/
  - http://www.statmt.org/moses/

A few others local search variants

Stochastic beam search

- Instead of choosing *k* best from the pool, choose *k* semi-randomly

Taboo list: prevent returning quickly to same state

- keep a fixed length list (queue) of visited states
- add most recent and drop the oldest
- never visit a state that's in the taboo list

Idea 5: genetic algorithms

We have a pool of *k* states

Rather than pick from these, **create** new states by combining states

Maintain a "population" of states

Genetic Algorithms

A class of probabilistic optimization algorithms

- A genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators

Inspired by the biological evolution process

Uses concepts of "Natural Selection" and "Genetic Inheritance" (Darwin 1859)

Originally developed by John Holland (1975)

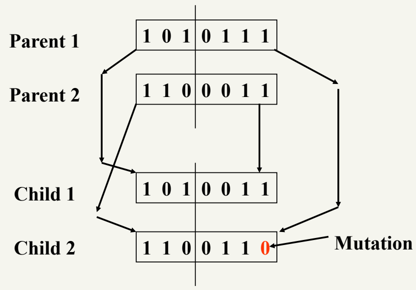
The Algorithm

Randomly generate an initial population.

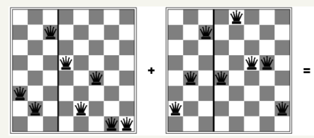
Repeat the following:

1. Select parents and "reproduce" the next generation
2. Randomly mutate some
3. Evaluate the fitness of the new generation
4. Discard old generation and keep *some* of the best from the new generation

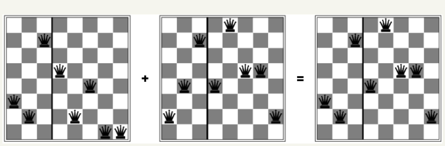
Genetic Algorithm Operators Mutation and Crossover



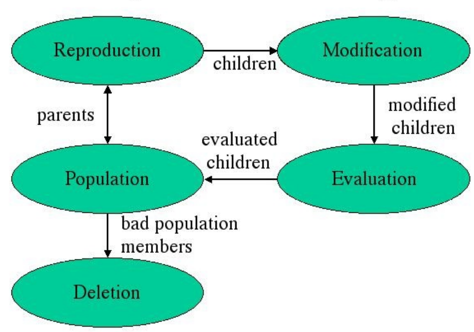
Genetic algorithms



Genetic algorithms



Anatomy of a Genetic Algorithm



Local Search Summary

Surprisingly efficient search technique

Wide range of applications

Formal properties elusive

Intuitive explanation:

- Search spaces are too large for systematic search anyway. . .

Area will most likely continue to thrive

Local Search Example: SAT

Many real-world problems can be translated into propositional logic:

(A v B v C) ^ (¬B v C v D) ^ (A v ¬C v D)

. . . solved by finding truth assignment to variables (A, B, C, . . .) that satisfies the formula

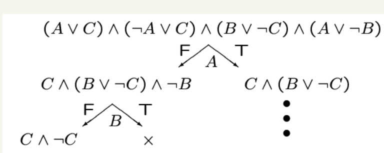
Applications

- planning and scheduling
- circuit diagnosis and synthesis
- deductive reasoning
- software testing
- . . .

Satisfiability Testing

Best-known systematic method:

- Davis-Putnam Procedure (1960)
- Backtracking depth-first search (DFS) through space of truth assignments (with unit-propagation)



Greedy Local Search (Hill Climbing)

Greedy Local Search (Hill Climbing): GSAT

GSAT:

1. Guess random truth assignment
2. Flip value assigned to the variable that yields the greatest # of satisfied clauses. (Note: Flip even if no improvement)
3. Repeat until all clauses satisfied, or have performed “enough” flips
4. If no sat-assign found, repeat entire process, starting from a different initial random assignment.

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

(Selman, Levesque, and Mitchell 1992)

GSAT vs. DP on Hard Random Instances

form.	GSAT			Davis-Putnam		
vars	m.flips	retries	time	choices	depth	time
50	250	6	0.5 *sec*	77	11	1 *sec*
70	350	11	1 *sec*	42	15	15 *sec*
100	500	42	6 *sec*	10^3	19	3 *min*
120	600	82	14 *sec*	10^5	22	18 *min*
140	700	53	14 *sec*	10^6	27	5 *hrs*
150	1500	100	45 *sec*	—	—	—
200	2000	248	3 *min*	—	—	—
300	6000	232	12 *min*	—	—	—
500	10000	996	2 *hrs*	10^{30}	> 100	10^{19} *yrs*

Notes: Define “Hard” later
Only “satisfiable” formulae
(else GSAT does not terminate)

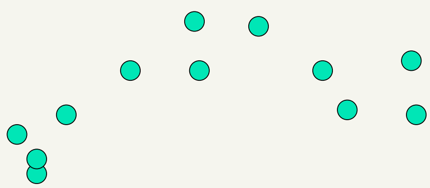
Experimental Results: Hard Random 3SAT

	GSAT				Simul. Ann.	
	basic		walk			
vars	time	eff.	time	eff.	time	eff.
100	.4	.12	.2	1.0	.6	.88
200	22	.01	4	.97	21	.86
400	122	.02	7	.95	75	.93
600	1471	.01	35	1.0	427	.3
800	*	*	286	.95	*	*
1000	*	*	1095	.85	*	*
2000	*	*	3255	.95	*	*

- Effectiveness: prob. that random initial assignment leads to a solution.
- Complete methods, such as DP, up to 400 variables
  - Mixed Walk better than Simulated Annealing
  - better than Basic GSAT
  - better than Davis-Putnam

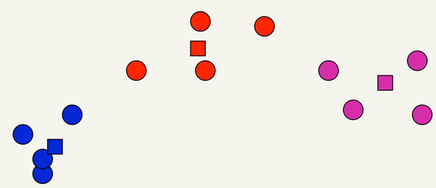
Local search for mancala?

Clustering



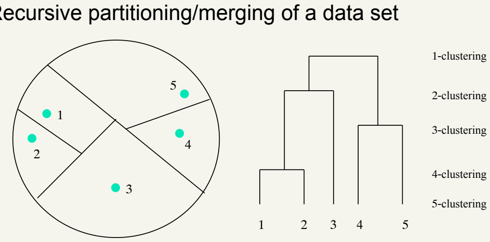
Group together similar items. Find clusters.

For example…



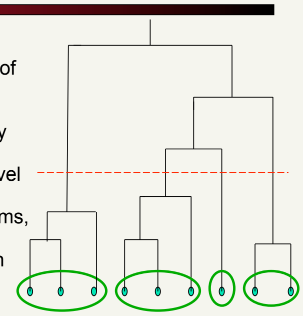
Hierarchical Clustering

Recursive partitioning/merging of a data set



Dendogram

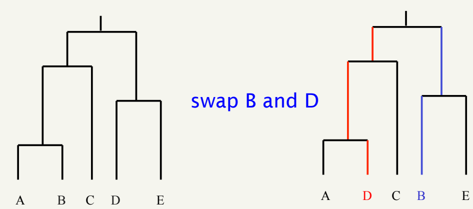
- Represents all partitionings of the data
- We can get a K clustering by looking at the **connected** components at any given level
- Frequently binary dendograms, but n-ary dendograms are generally easy to obtain with minor changes to the algorithms



Hierarchical clustering as local search

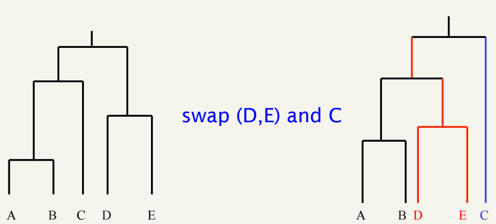
- State?
  - a hierarchical clustering of the data
  - basically, a tree over the data
  - huge state space!
- "adjacent states"?
  - swap two sub-trees
  - can also "graft" a sub-tree on somewhere else

Swap without temporal constraints, example 1



no change to the structure

Swap without temporal constraints, example 2



structure changed!

Hierarchical clustering as local search

- state criterion?

Hierarchical clustering as local search

- state criterion?
  - how close together are the k-clusterings defined by the hierarchical clustering

$$\text{hcost} = \sum_{i=1}^{n} w_k \,\text{cost}(C_k)$$

weighted mean of k-clusterings

$$\text{cost}(C_k) = \sum_{j=1}^{k} \sum_{x \in S_j} \left\| x - \mu(S_j) \right\|^2$$

sum of squared distances from cluster centers

SS-Hierarchical vs. Ward's

Yeast gene expression data set

	SS-Hierarchical Greedy, Ward's initialize	Ward's
20 points	21.59 8 iterations	21.99
100 points	411.83 233 iterations	444.15
500 points	5276.30 ? iterations	5570.95