

Iterative Improvement Search

with several slides inspired by/drawn from Dan Klein, Stuart Russell, Andrew Moore

CSC384 Summer 2019

What's Iterative Improvement Search?

A form of search that starts with a proposed solution and iterates to make it better. Can be used when:

- States can be represented as combinatorial structures to be optimized.
- There is a cost function (or objective function) to map structures (or states) onto real numbers.
- The function makes similarly “good” solutions have similar costs.
- Searching all possible combinations is intractable.
- Depth first search approaches are too expensive.
- There's no known algorithm for finding the optimal solution efficiently.
- You don't care how you arrive at a solution, you just want to find a solution.

Iterative Search

Imagine all of the state configurations laid out on the surface of a landscape that is organized by the objective function.

Iterative improvement search will want to find the highest point on the landscape.

Unlike other AI search problems like 8-puzzle, we don't care how we get to this point.

1. Start at a random configuration;
2. Repeatedly consider various moves;
3. Accept some, reject some;
4. When you get stuck, you restart.

The algorithms require a moveset (or successor function) that describes the moves we can consider from any given configuration.

What's a plausible objective function and moveset for 8 puzzle?

What's a plausible objective function and moveset for Sudoku?

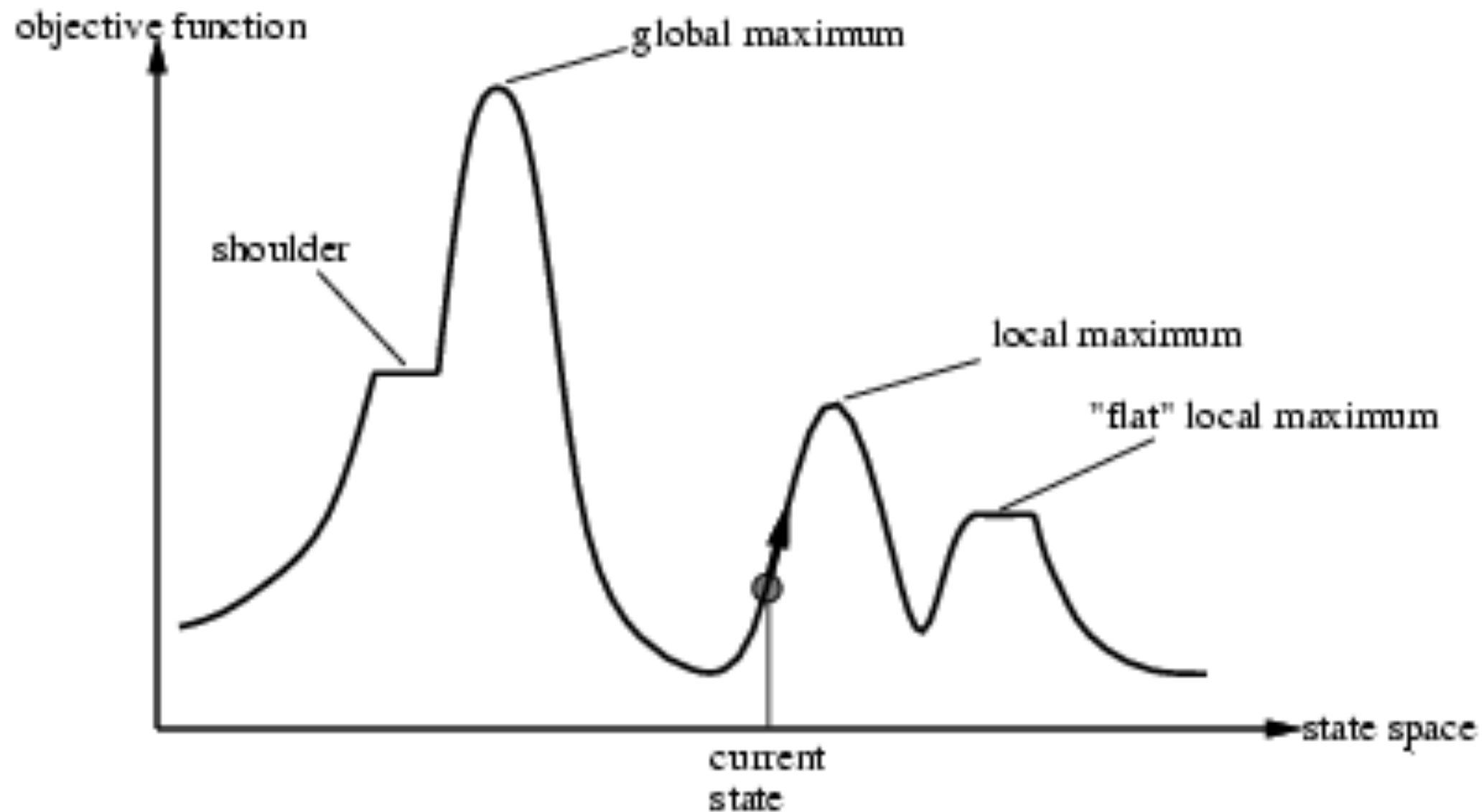
Hill-climbing search

"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                  neighbor, a node

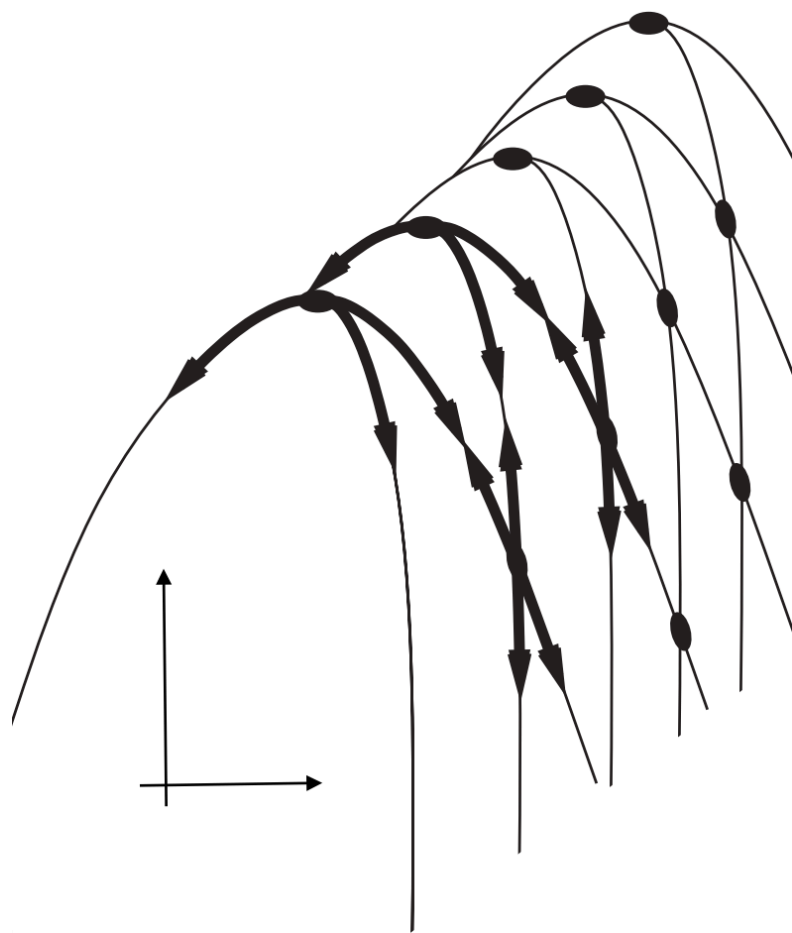
  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```

Hill-climbing landscapes



Hill-climbing landscapes

Diagonal ridges:



From each local maximum all the available actions point downhill, but there is an uphill path.

May result in a zig-zag motion through the search space and very long ascent time!

If you can compute the gradient of our objective function we can avoid this issue: this is the idea behind **gradient descent**.

Solving Sudoku by Hill Climbing

- An objective function can be given by the number of constraint violations among numbers assigned to the grid.
- We can search for states (or configurations of numbers) that result in a minimum number of constraint violations.

This is called the min-conflicts algorithm

Min-conflicts and n-Queens

- Put n queens on an $n \times n$ board such that no two queens are on the same row, column, or diagonal.



Min-conflicts and n-Queens

State: Position of the n queens, one per column (or row)

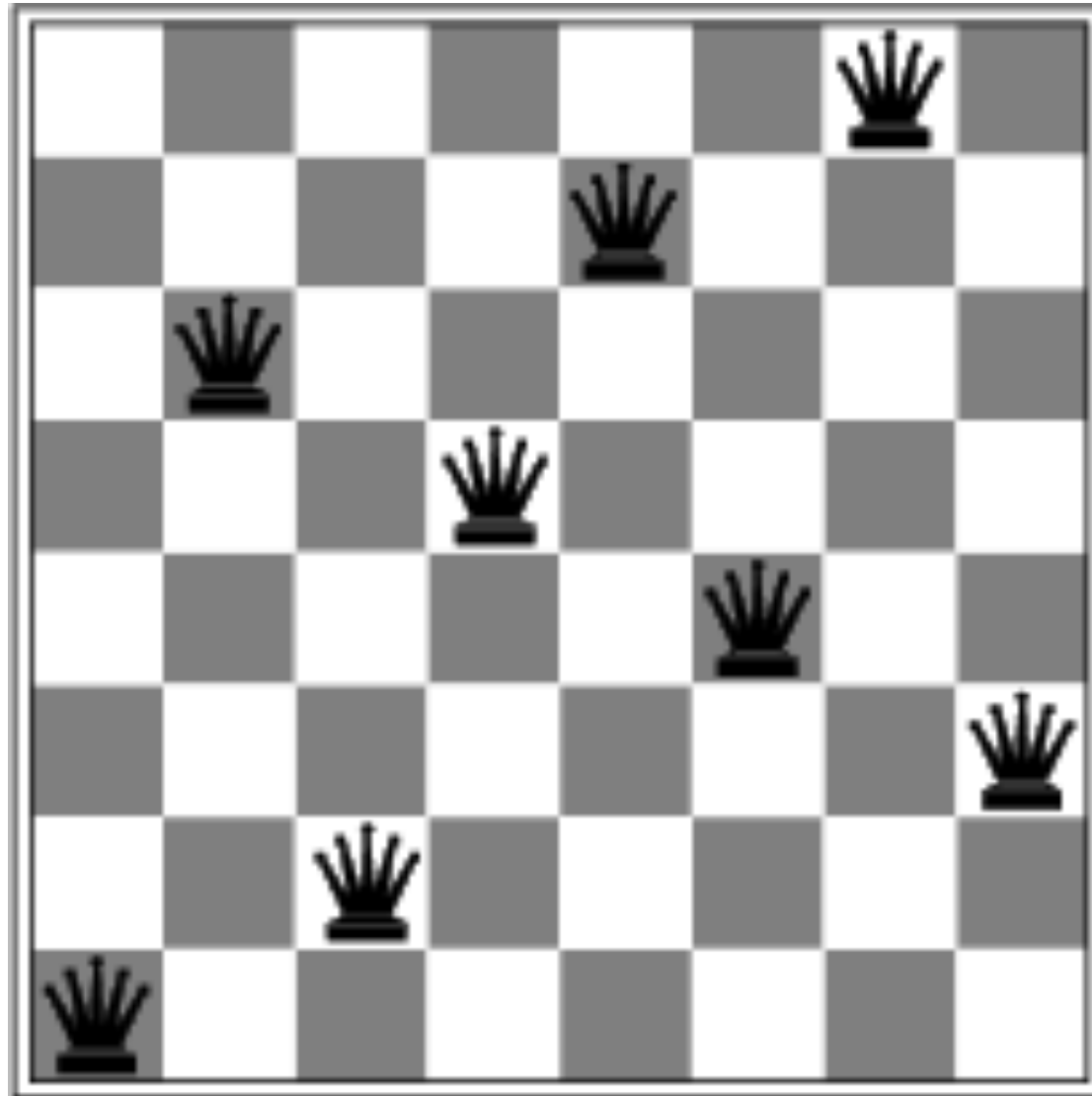
Moveset (or successors):
generated by moving a single queen to another square in its column (size of this set is $n(n-1)$)

Objective function ($f(\text{state})$):
the number of constraint violations

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♙	13	16	13	16
♙	14	17	15	♙	14	16	16
17	♙	16	18	15	♙	15	♙
18	14	♙	15	15	14	♙	16
14	14	13	17	12	14	12	18

- $f(\text{state}) = \#$ of pairs of queens that are attacking each other
- $f(\text{state}) = 17$ for the above state

Min-conflicts and n-Queens



A local minimum with $h = 1$

Min-conflicts and n-Queens

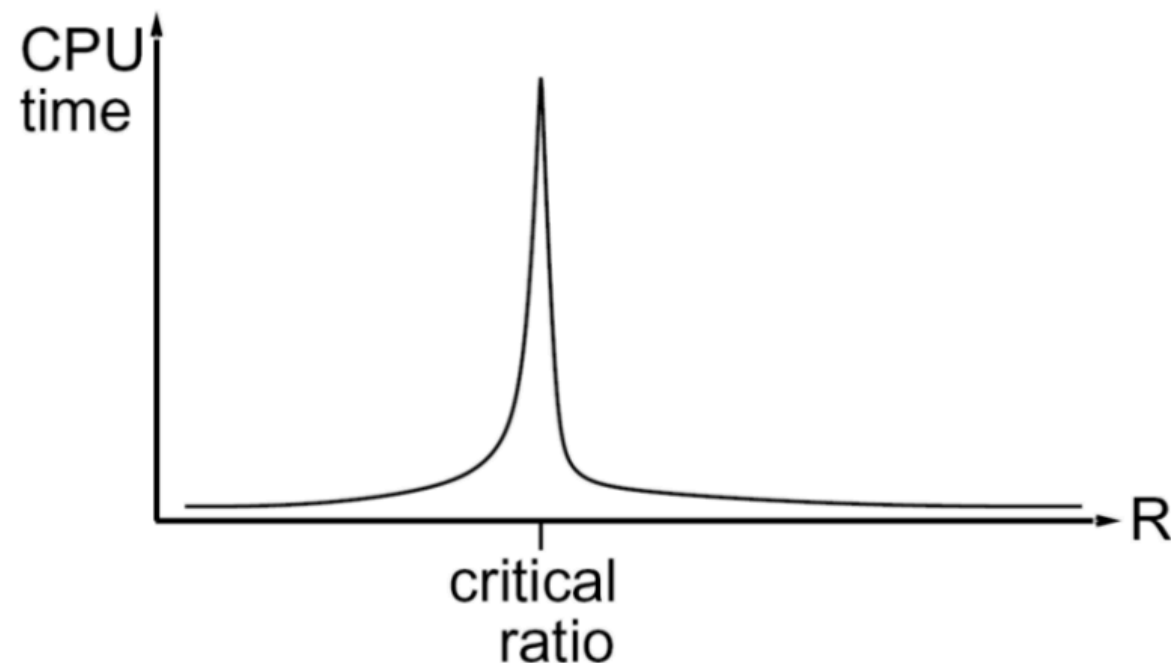
8 queens statistics:

- State space includes 8^8 or about 17 million states.
- Starting from random state, steepest-ascent hill climbing solves 14% of problem instances.
- 86% of the time it gets stuck in a local minima, tho.
- It takes 4 steps on average when it succeeds, 3 when it gets stuck.
- When random restarts are allowed, performance improves.

Min-conflicts and n-Queens

Curiously, given a random initial state, the algorithm can solve large n-queens problems in almost constant time for arbitrary n.

$$R = (\# \text{ of constraints})/(\# \text{ of variables})$$



Watch min-conflicts on a map colouring problem:
<https://www.youtube.com/watch?v=QGH9g-CNPxQ>

Hill-climbing search

The good:

Easy to implement

Can perform well also in large (infinite, continuous) spaces

Can terminate anytime to yield solution, sometimes this solution will be “good enough”

Must remember only current state and potential successors

The bad:

Not complete (nor optimal)

If the number of moves is enormous, the algorithm may be inefficient. What to do?

If the number of moves is tiny, the algorithm can get stuck easily. What to do?

Considerations:

1. MoveSet design is critical as is the design of the objective function.
2. Problems: dense local optima or plateaux
3. It's often cheaper to evaluate an incremental change of a previously evaluated configuration than to evaluate from scratch. Does hill-climbing permit that?

Improving hill-climbing

Sideways moves: if there are no uphill moves, allow moves to a state with the same value as the current one (to escape ‘shoulders’).

When 100 of such moves are allowed in sequence for the 8-queens problem, solutions are found 94% (instead of 14%!) of the time.

Stochastic hill-climbing: select randomly among available uphill moves. Note that a ‘random walk’ selects among all moves at random.

Random-restart hill climbing: restart at a random location. This can be demonstrated “probabilistically” complete.


Restarts	0	2	4	8	16	32	64
Success?	14%	36%	53%	74%	92%	99%	99.994%

First-choice hill-climbing: generate successors randomly, one at a time, until one that is better than the current state is found (to deal with large movesets).

Tabu list: Maintain a list of 100 or so visited states, so that we don’t return to these (to escape ‘plateaux’).

Moveset Examples: WALKSAT

$A \vee \neg B \vee C$	1	Maximize: Eval(config) = # of satisfied clauses	Moveset: flip any 1 variable	Example Configuration: (1,0,1,0,1)
$\neg A \vee C \vee D$	1			
$B \vee D \vee \neg E$	0			
$\neg C \vee \neg D \vee \neg E$	1			
$\neg A \vee \neg C \vee E$	1			

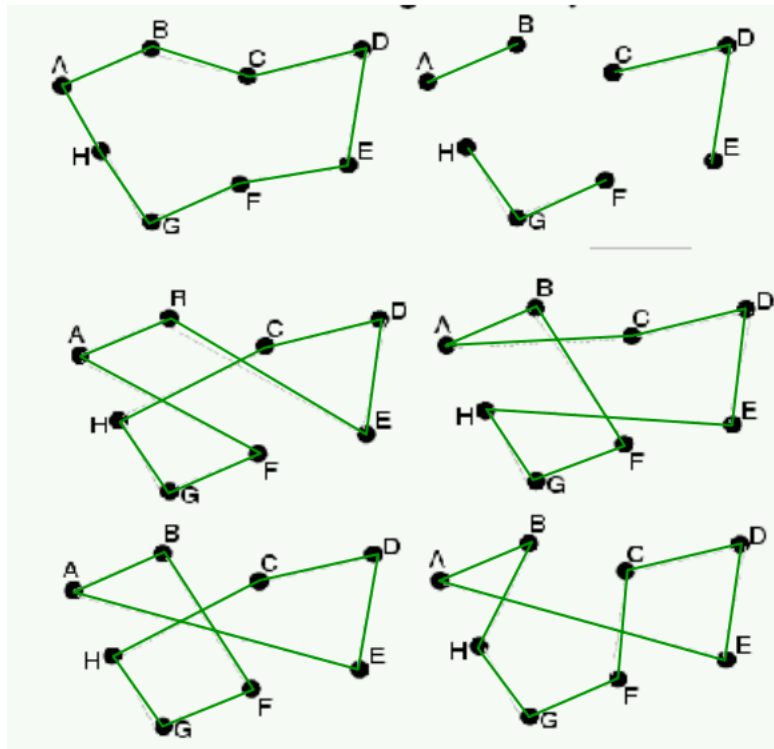


Algorithm:

1. Pick a random unsatisfied clause
2. Consider 3 moves: one that flips each variable.
3. If any yield improved results, accept the best. If none improve results, then 50% of the time pick the move that is the “least” bad and 50% of the time pick one at random.

*This is the best known algorithm for satisfying Boolean formulae
[Selman]*

Moveset Examples: Travelling Salesperson Problem



Minimize the distance required to “tour” of all the vertices.

Movesets here may change two edges at a time, three edges or more (“3-changes” are illustrated at left).

Empirically, author Lin has found:

- 3-change solutions are better than 2-change ones.
- 4-change solutions are not sufficiently better than 3-change ones to justify the extra compute time.
- A 3-change tour for a 48-city problem from authors Held and Karp was found to have a probability of about 0.05 of being optimal (meaning 100 random restarts will yield the optimal solution with probability 0.99).
- *There is no theoretical justification for any of these results.*

Simulated annealing search

Idea: make the search fall out of mediocre local minima and into better local maxima by “shaking” the landscape, i.e. adding some probabilistic moves.

But how much to “shake”?

- **Idea One:** Accept a new position with probability = 0.1
- **Idea Two:** Accept a new position with probability that decreases with time.
- ***Idea Three:*** Accept a new position with probability that decreases with time and as the change in objective function increases.

Simulated annealing search

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state

inputs: *problem*, a problem

schedule, a mapping from time to “temperature”

local variables: *current*, a node

next, a node

T, a “temperature” controlling prob. of downward steps

current \leftarrow MAKE-NODE(INITIAL-STATE[*problem*])

for *t* \leftarrow 1 **to** ∞ **do**

T \leftarrow *schedule*[*t*]

if *T* = 0 **then return** *current*

next \leftarrow a randomly selected successor of *current*

$\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]

if $\Delta E > 0$ **then** *current* \leftarrow *next*

else *current* \leftarrow *next* only with probability $e^{\Delta E/T}$

Simulated annealing search

Note that $e^{-\Delta E/T}$ defines a *Boltzman distribution*

T is a “temperature” parameter that gradually decreases at each iteration.

High temp: Accept all moves

Low temp: Stochastic Hill-Climbing

When enough iterations have passed without improvement, terminate. This idea was introduced by Metropolis in 1953. It is based on the way that alloys manage to find a nearly global minimum energy level when they are cooled slowly.

Simulated annealing search

The good:

Sometimes empirically much better at avoiding local minima than hill-climbing. It is a successful, frequently-used, algorithm.

The bad:

Can't say much formal about it.

But one can prove: If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1.

Considerations:

1. MoveSet design is critical as is the design of the objective function.
2. Annealing schedule is critical.
3. It's often cheaper to evaluate an incremental change of a previously evaluated configuration than to evaluate from scratch. Does simulated annealing permit that?

Beam search

Idea:

Keeping only one configuration in memory at a time is rather extreme. Why not keep many configurations in memory?

Algorithm outline:

1. *select k states at random*
2. *determine all successors of the k states*
3. *terminate when maxima is located, else*
4. *select k best from successors and repeat*

Beam search

The good:

Searches that find good states recruit others to join them.

The bad:

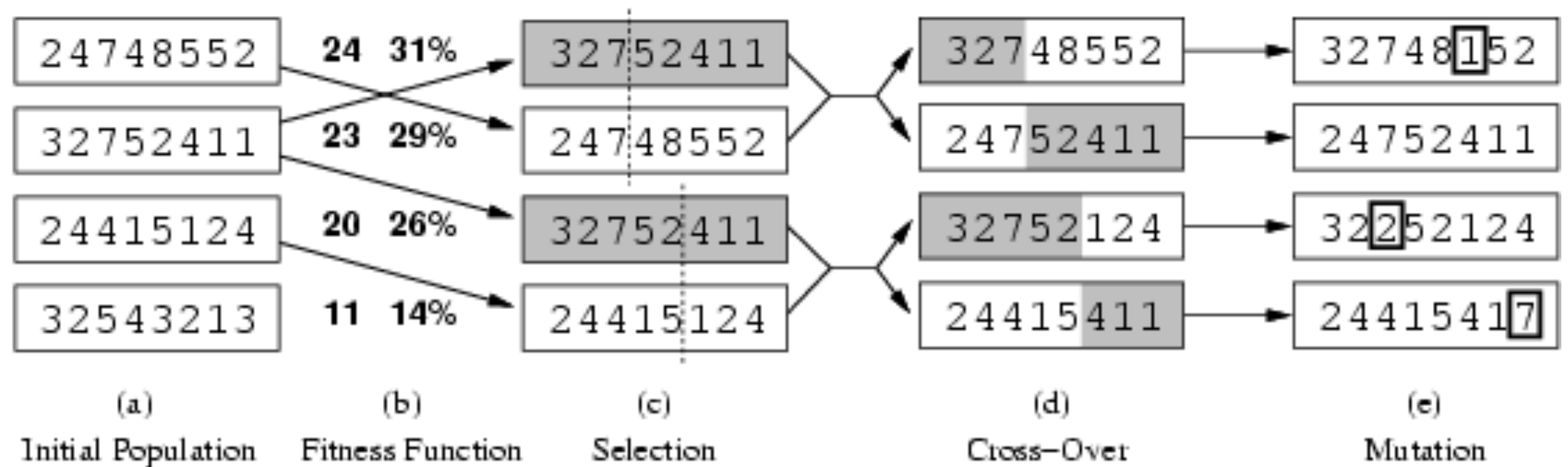
All states in the sample set may end up lying on the same hill.

Randomization in selection of successors can be introduced to ameliorate this problem.

Genetic algorithms

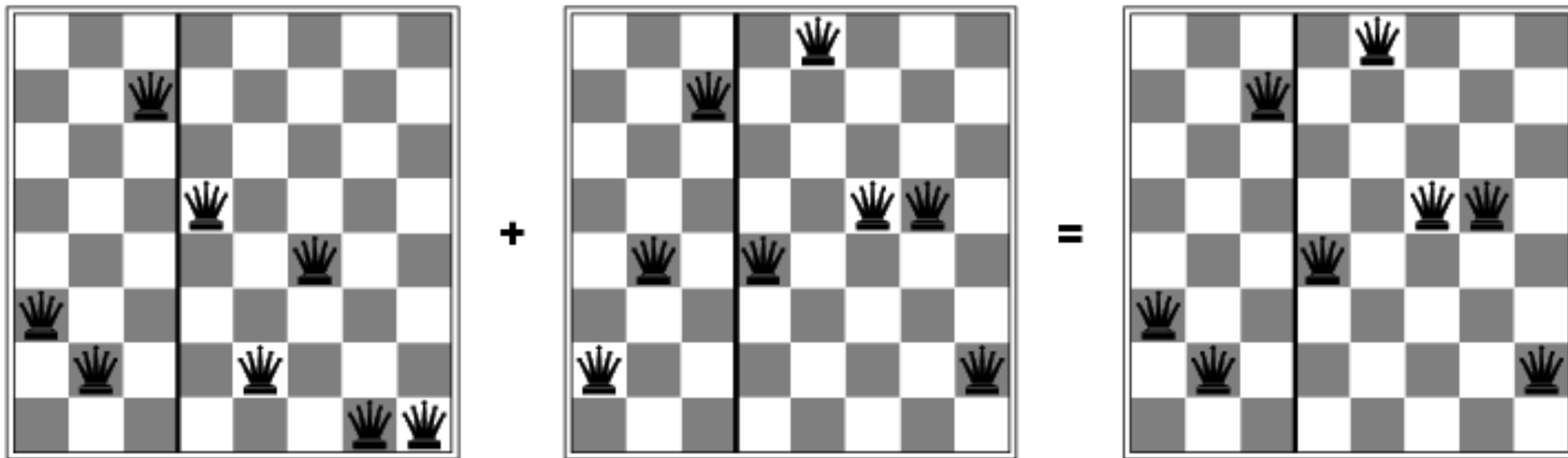
- Similar to stochastic beam search.
- Start with k randomly generated states (a **population**) and an objective function (**fitness function**).
- At each iteration, produce the next generation of states by **selection, crossover, and mutation**

Genetic algorithms



- Objective (or fitness) function: the number of non-attacking pairs of queens.
- Percentages illustrate relative fitness, e.g. $24/(24+23+20+11) = 31\%$. These percentages define the probability of being sampled in the next generation.
- Cross-overs have the effect of “jumping” to completely different parts of the search space (quite non-local).
- Mutations are relatively local.

Genetic algorithms: Crossover



Genetic algorithms

Elements in each generation are a set of strings.

Algorithm:

Start with a generation G of N random strings ($b_0, b_0 \dots b_{N-1}$).

For each string b_i , define $p_i = \text{Eval}(b_i) / \sum_j \text{Eval}(b_j)$

Let G' be the next generation. Begin with it empty.

For $k = 0 ; k < N/2 ; k = k+1$:

- Choose two parents each with probability $\text{Prob}(\text{Parent} = b_i) = p_i$
- Randomly swap elements in the two parents to obtain two new strings (i.e. apply crossovers)
- For each element in the new strings, randomly invert with some low probability (i.e. add a mutation)
- Add the two new strings to G' and repeat

Genetic algorithms

The good:

Easy to implement

Random exploration can find solutions that local search can't (via crossover)

Appealing connection to human evolution

The bad:

Large number of “tunable” parameters (i.e. crossover designs, mutation probabilities)

Difficult to replicate performance from one problem to another

Lack of empirical studies comparing algorithms to simpler methods

No convincing evidence that they outperform hill-climbing w/random restarts, in general

Often the “second best way” to solve a problem

Considerations:

String representation critical, nuanced (how might you encode TSP?)

Objective (fitness) functions critical