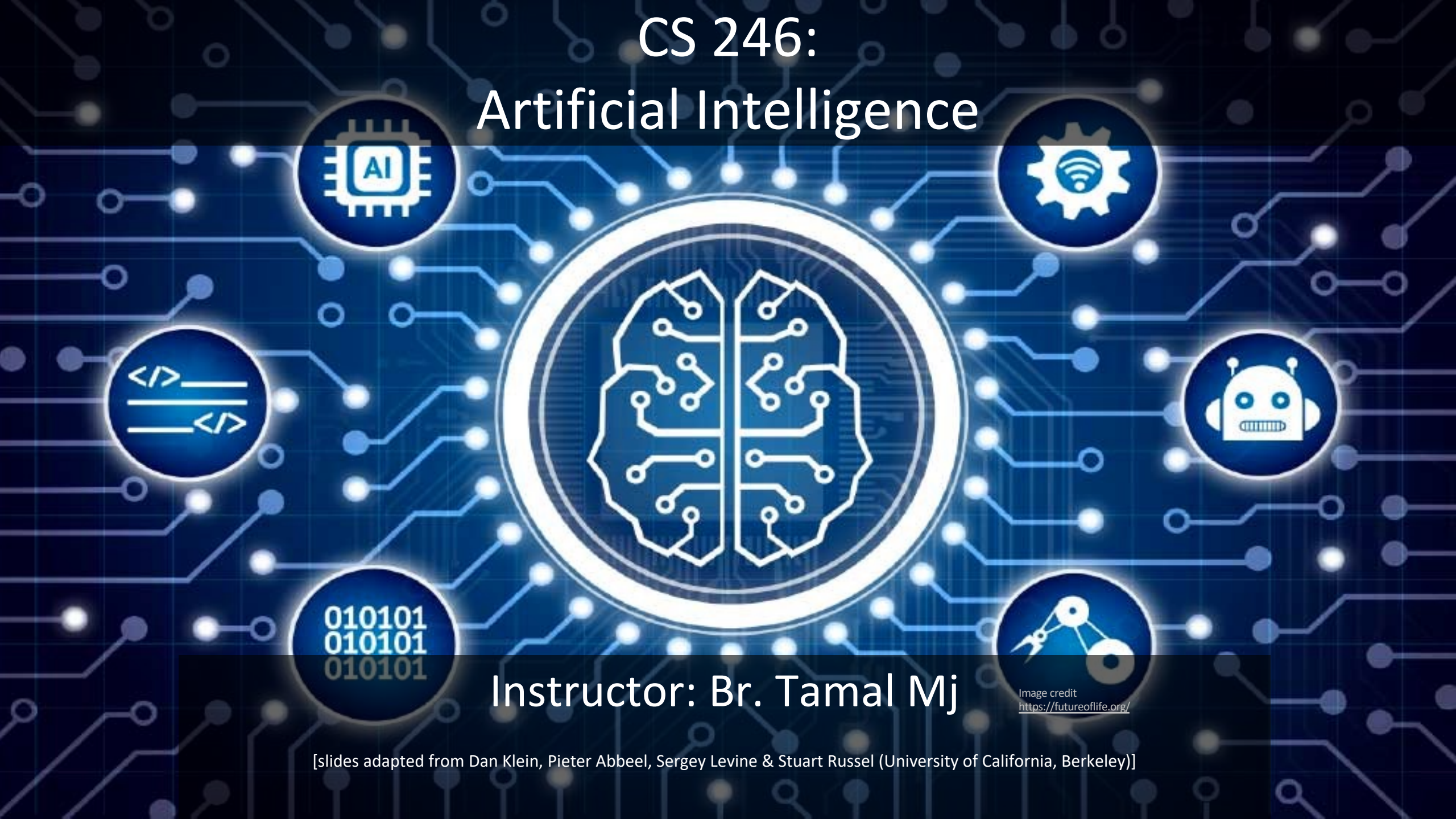
CS 246:
Artificial Intelligence



Instructor: Br. Tamal Mj

Image credit
https://futureoflife.org/

[slides adapted from Dan Klein, Pieter Abbeel, Sergey Levine & Stuart Russel (University of California, Berkeley)]

Om Saha Naav[au]-Avatu
Saha Nau Bhunaktu
Saha Viiryam Karavaavahai
Tejasvi Naav[au]-Adhiitam-
Astu Maa Vidvissaavahai
Om Shaantih Shaantih
Shaantih

Om, May we all be protected
May we all be nourished
May we work together with great energy
May our intelect be sharpened (may our study be effective)
Let there be no Animosity amongst us
Om, peace (in me), peace (in nature), peace (in divine forces)

Local Search

Traditional Search Algorithms

- Traditional search algorithms explore search spaces systematically.
- Systematicity is achieved by:
  - Keeping one or more paths in memory.
  - Recording explored alternatives at each point along the path.
- When a goal is found, the path to the goal constitutes a solution.
- In some problems, the path to the goal is irrelevant
  - e.g., 8-queens problem, Vehicle routing, Integrated-circuit design.
- In such cases, the final configuration is more important than the order of actions.

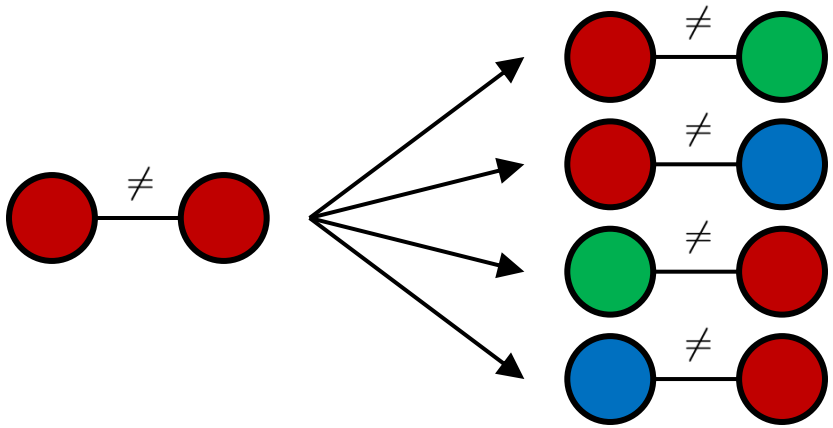
Local Search Algorithms

- Suitable when the path does not matter.
- Characteristics of local search algorithms:
  - Use a single current node, not multiple paths.
  - Move only to neighbors of the current node.
  - Paths followed are typically not retained.
- Advantages of local search algorithms:
  - Use very little memory, usually a constant amount.
  - Can find reasonable solutions in large or infinite state spaces.
  - Suitable for problems where systematic algorithms are unsuitable.
  - Useful for solving optimization problems (eg. Gradient Descent)

Local Search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)

- Local search: improve a single option until you can't make it better
- New successor function: local changes



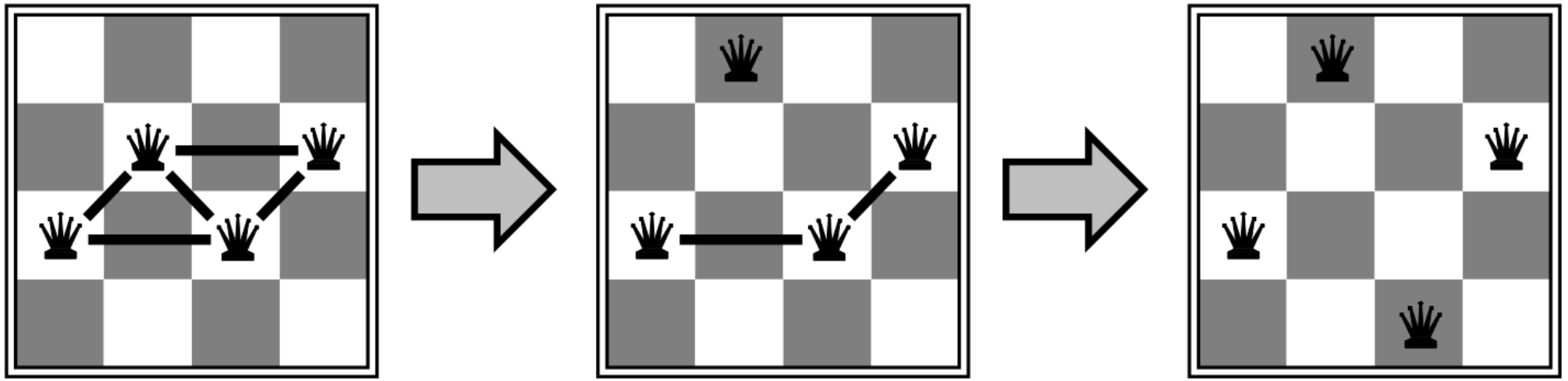
- Generally much faster and more memory efficient (but incomplete and suboptimal)

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

Heuristic for *n*-queens problem

- Goal: n queens on board with no ***conflicts***, i.e., no queen attacking another
- States: n queens on board, one per column
- Heuristic value function: number of conflicts

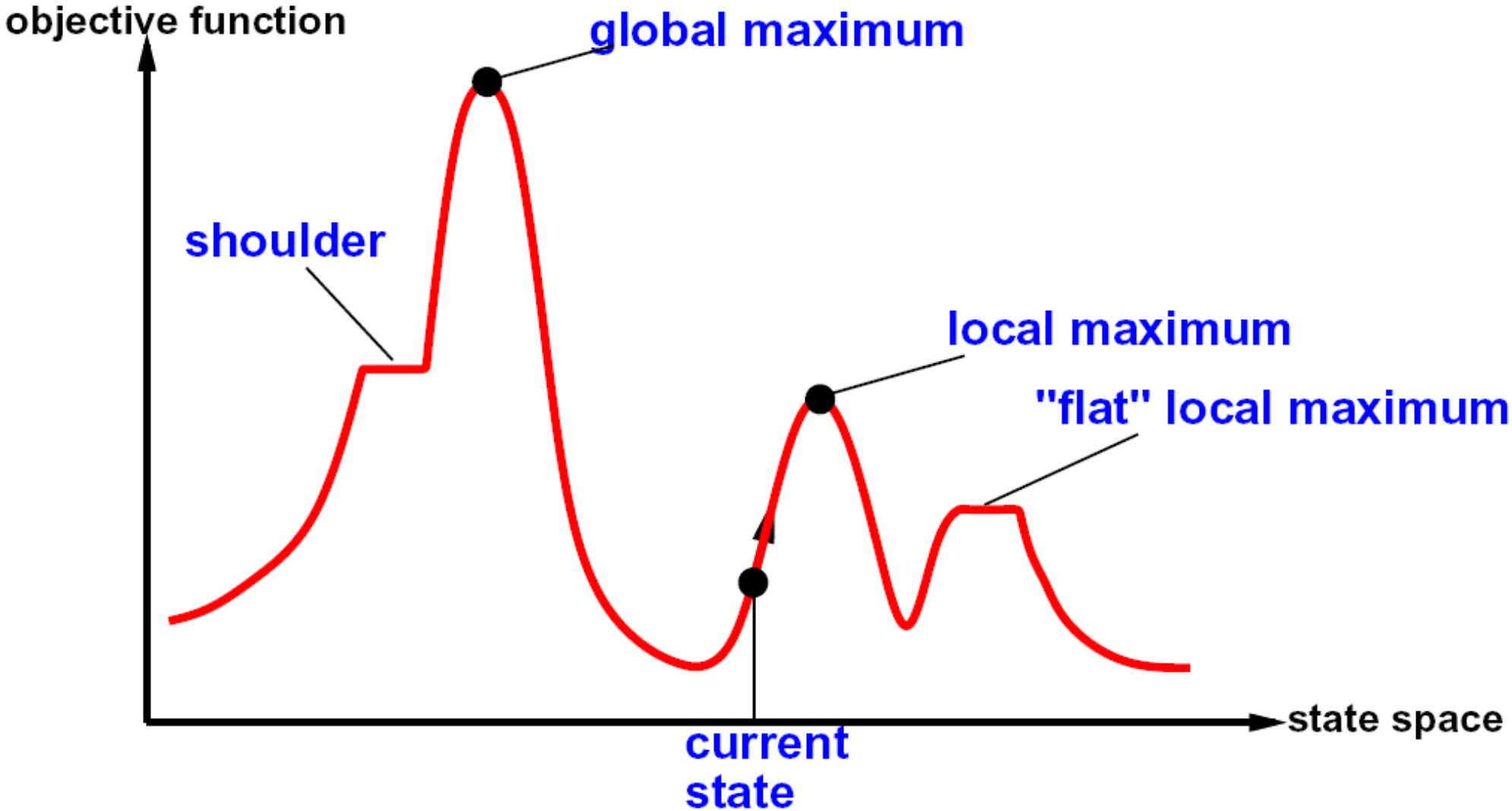


Hill Climbing

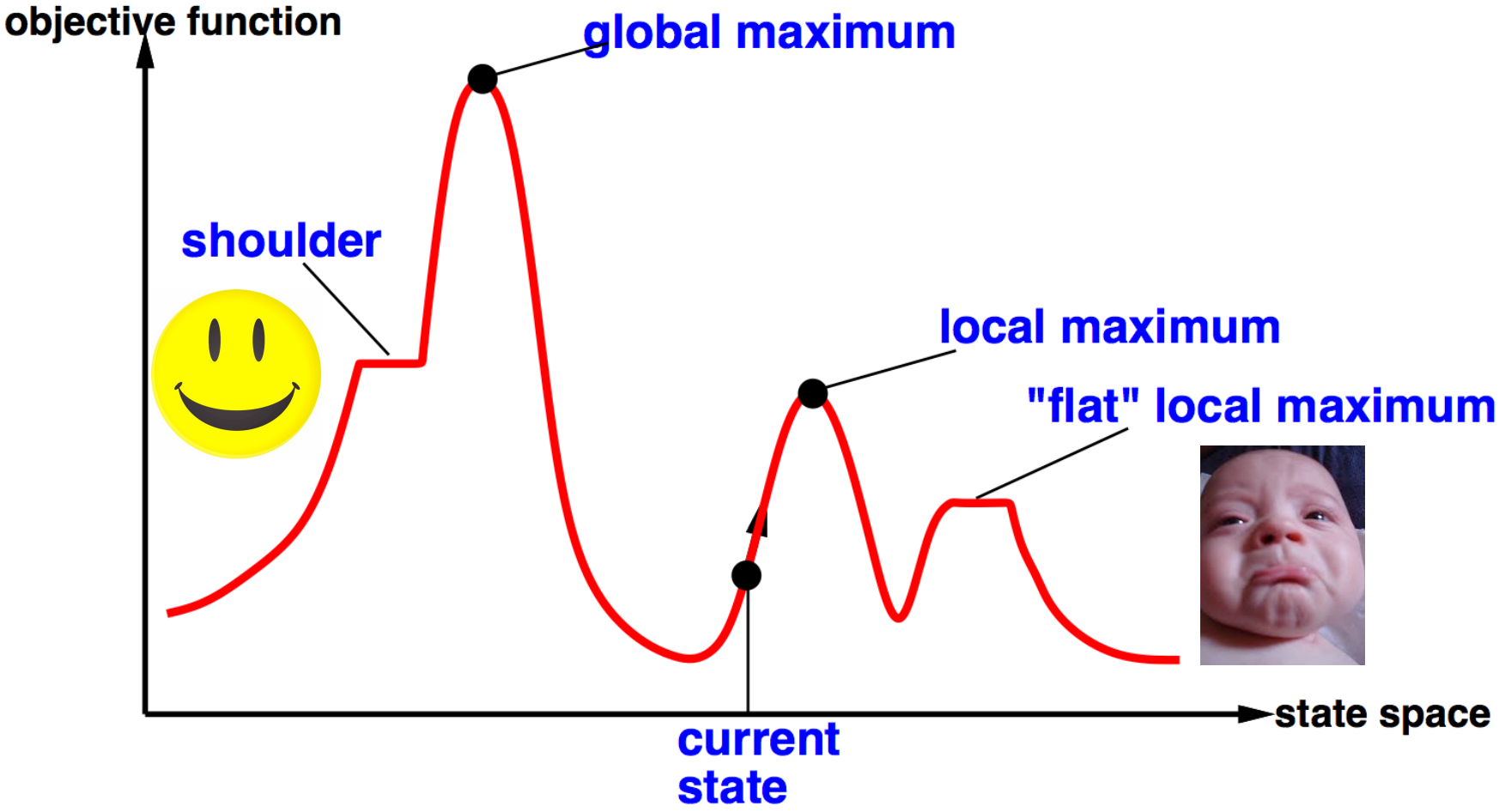
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    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
    end
```

Hill Climbing Diagram



Global and local maxima



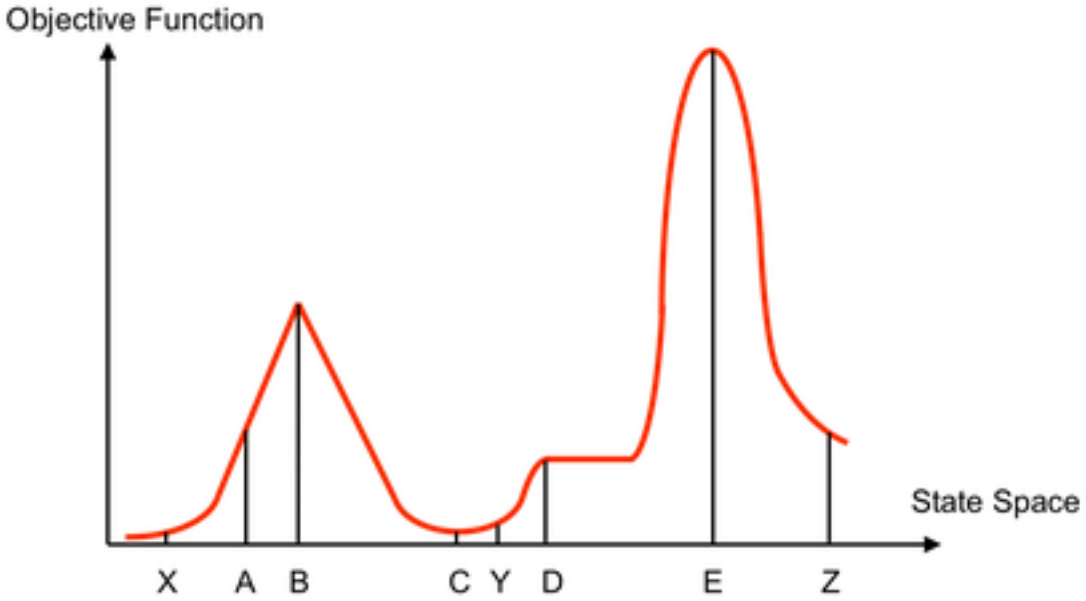
Random restarts

- find global optimum
- duh

Random sideways moves

- Escape from shoulders
- Loop forever on flat local maxima

Hill Climbing Quiz



Starting from X, where do you end up ?

Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

Hill Climbing Variants

- Stochastic hill climbing:
  - chooses at random from among the uphill moves;
  - the probability of selection can vary with the steepness of the uphill move.
  - This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
- First-choice hill climbing:
  - implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
  - This is a good strategy when a state has many (e.g., thousands) of successors

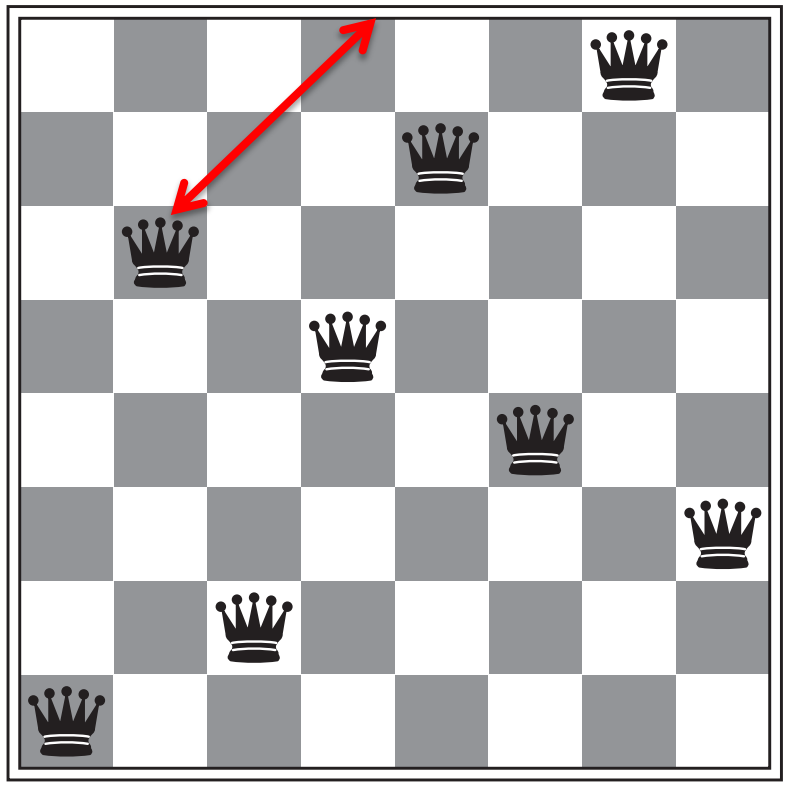
Random-Restart Hill Climbing

- These hill-climbing algorithms are incomplete
  - Often fail to find a goal when one exists because they can get stuck on local maxima.
  - Solution: Random-Restart Hill Climbing
  - Adopt the well-known adage, “If at first you don’t succeed, try, try again.”
- Random-Restart Hill Climbing:
  - coducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.
  - It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state

Random Restart Hill-climbing

(on the 8-queens problem)

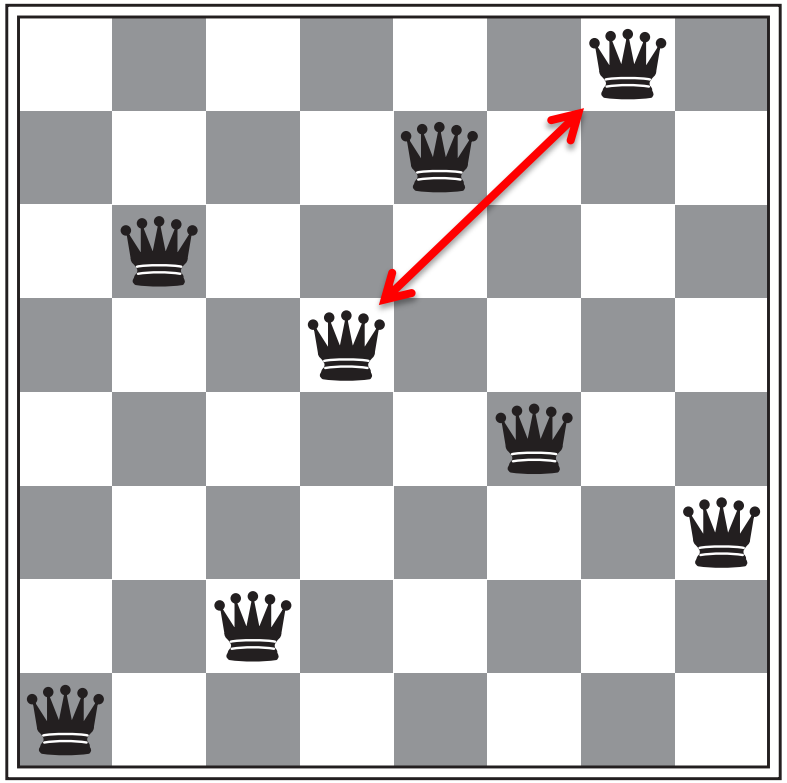
- No sideways moves:
  - If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$.
  - For 8-queens instances with no sideways moves allowed, $p \approx 0.14$
    - so we need roughly 7 iterations to find a goal (6 failures and 1 success).
  - Average number of moves per trial:
    - 4 when succeeding, 3 when getting stuck
  - Expected total number of moves needed:
    - $3(1-p)/p + 4 = \sim 22$ moves



Random Restart Hill-climbing

(on the 8-queens problem)

- Allowing 100 sideways moves:
  - Succeeds with prob. $p = 0.94$
  - Average number of moves per trial:
    - 21 when succeeding, 65 when getting stuck
  - Expected total number of moves needed:
    - $65(1-p)/p + 21 = \sim 25$ moves



For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.

Simulated Annealing

- A hill-climbing algorithm:
  - Never makes “downhill” moves toward states with lower value (or higher cost) ; can get stuck on a local maximum.
  - Efficient but incomplete
- A purely random walk:
  - Moves to a successor chosen uniformly at random from the set of successors
  - Complete but inefficient
- Simulated Annealing:
  - combines these two for completeness and efficiency

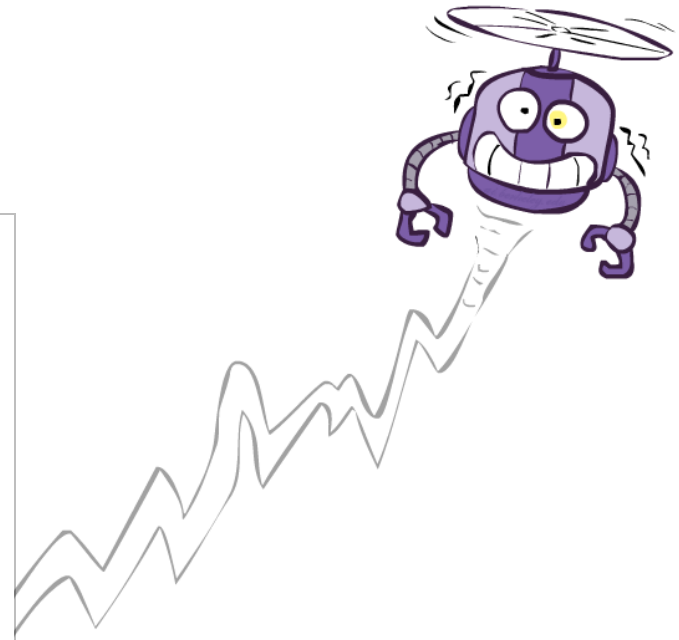
Annealing (in Metallurgy vs in AI)

- Annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.
- To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface.
- If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum.

- The trick:
  - Shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum.
  - The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature T) and then gradually reduce the intensity of the shaking (i.e., lower the temperature T).

Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on



```
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 ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```

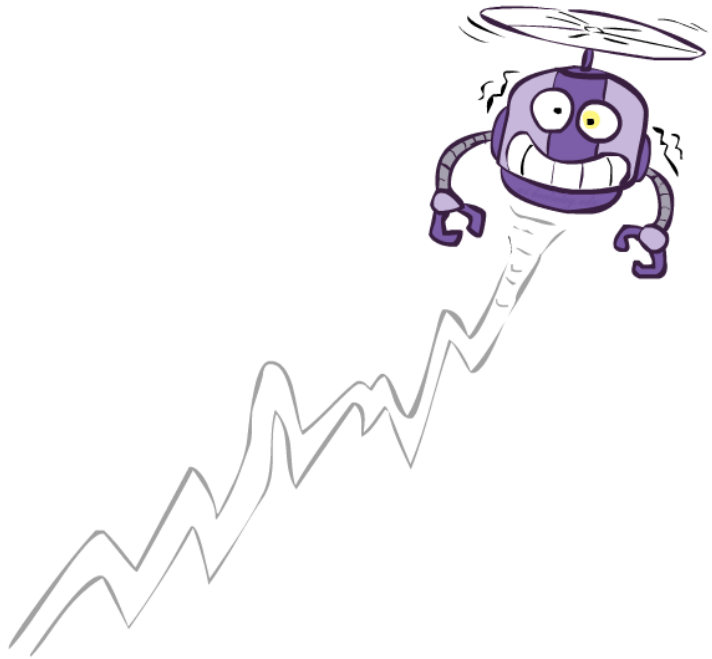
Instead of picking the best move, pick a random move

If the move improves the situation, it is always accepted.

Else, the algorithm accepts the move with prob. $p < 1$.
p decreases exponentially with the "badness" of the move.
p also decreases as the "temperature" T goes down

Simulated Annealing

- Theoretical guarantee:
  - Stationary distribution (Boltzmann): $p(x) \propto e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough,
    will converge to optimal state!



- Is this an interesting guarantee?

- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - "Slowly enough" may mean exponentially slowly
  - Random restart hillclimbing also converges to optimal state…

Local Beam Search

- Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations.
- The local beam search algorithm keeps track of k states rather than just one.
- It begins with k randomly generated states.
- At each step, all the successors of all k states are generated.
  - If any one is a goal, the algorithm halts.
  - Otherwise, it selects the k best successors from the complete list and repeats.

Local Beam Search vs. k random restarts in parallel

- The two algorithms are quite different.
  - In a random-restart search, each search process runs independently of the others.
  - In a local beam search, useful information is passed among the parallel search threads.
  - In effect, the states that generate the best successors say to the others, “Come over here, the grass is **greener**!”
  - The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

Stochastic Beam Search

- Local beam search can suffer from a lack of diversity among the k states
  - They can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing.
  - A variant called stochastic beam search helps alleviate this problem.
- Stochastic Beam Search:
  - Instead of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

Stochastic Beam Search → Natural Selection

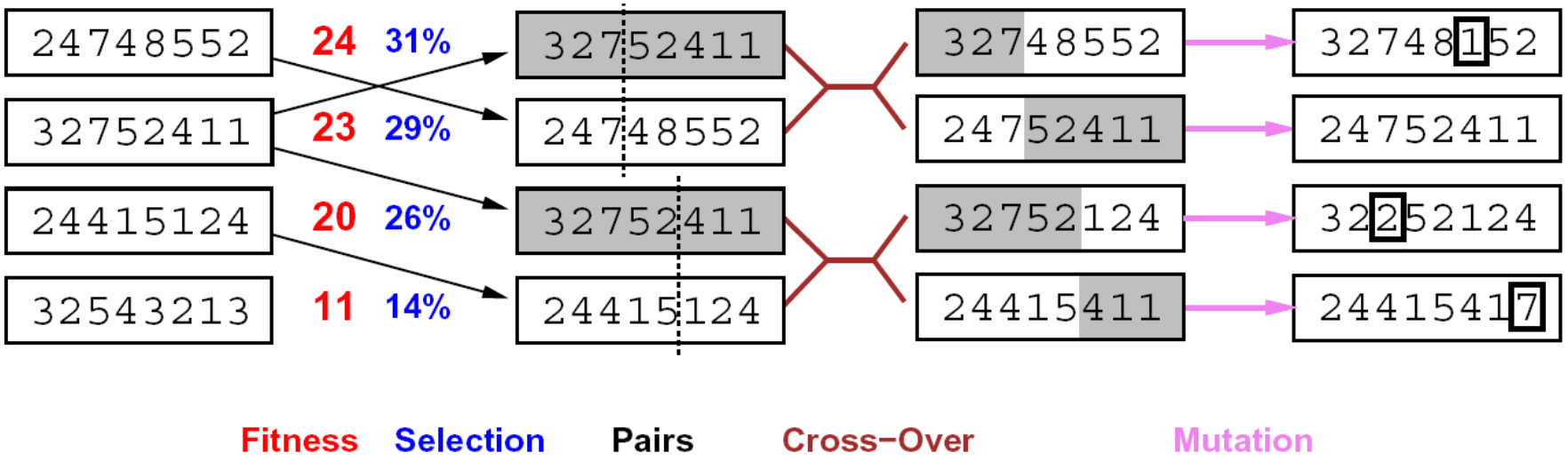
- Stochastic beam search bears some resemblance to the process of natural selection,
  - The “successors” (offspring) of a “state” (organism) populate the next generation according to its “value” (fitness).

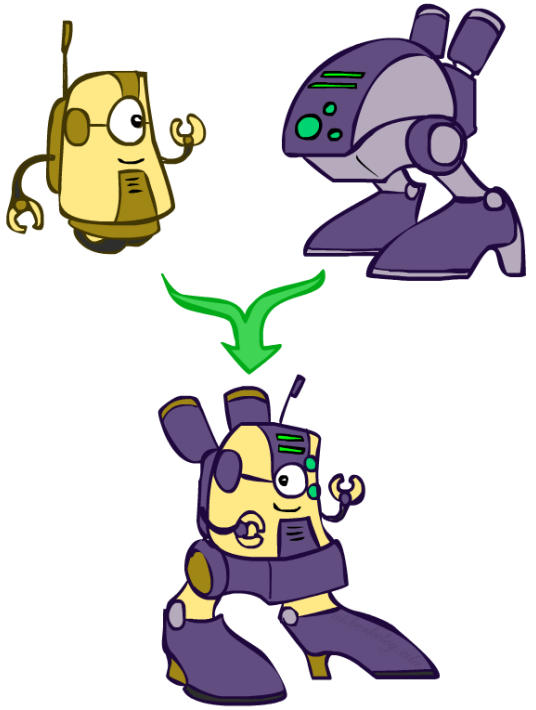
- This lead us to Genetic Algorithm

Genetic Algorithms

- A genetic algorithm (or GA) is a variant of stochastic beam search
  - successor states are generated by combining two parent states rather than by modifying a single state.
  - The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction.
  - Here the offspring (successor states) are produced through the combination of genetic material from two parents, inheriting characteristics from both parent states, creating diversity and potentially leading to better solutions.

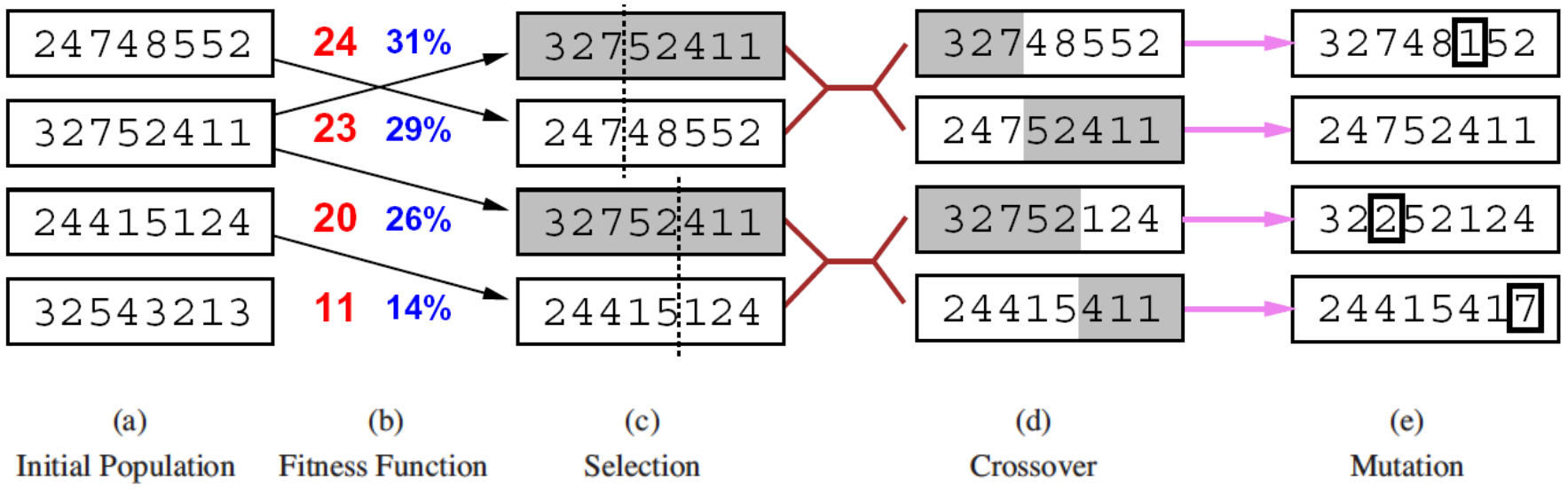
Genetic Algorithms





- Genetic algorithms use a natural selection metaphor
  - Keep best N hypotheses at each step (selection) based on a fitness function
  - Also have pairwise crossover operators, with optional mutation to give variety
- Possibly the most misunderstood, misapplied (and even maligned) technique around

Genetic Algorithms

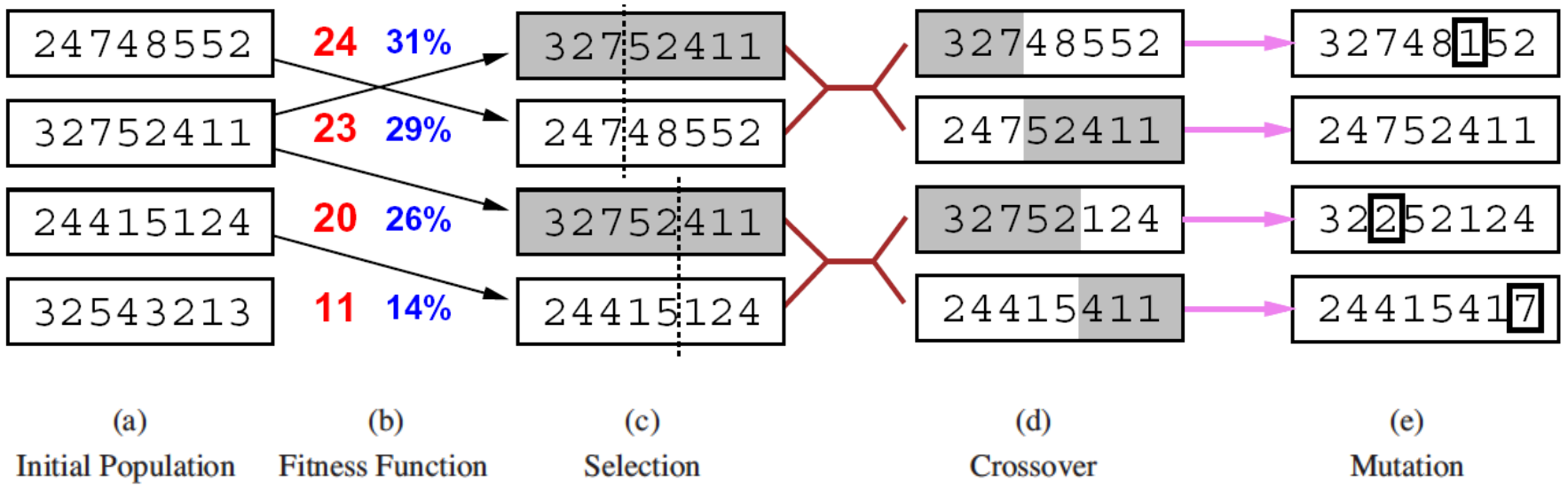


The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Genetic Algorithms

- Like beam searches, GAs begin with a set of k randomly generated states, called the population.
- Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s.
  - Eg: an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 * \log_2 8$ =24 bits.
  - Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8.
  - Figure shows a population of four 8-digit strings representing 8-queens states.

Genetic Algorithms

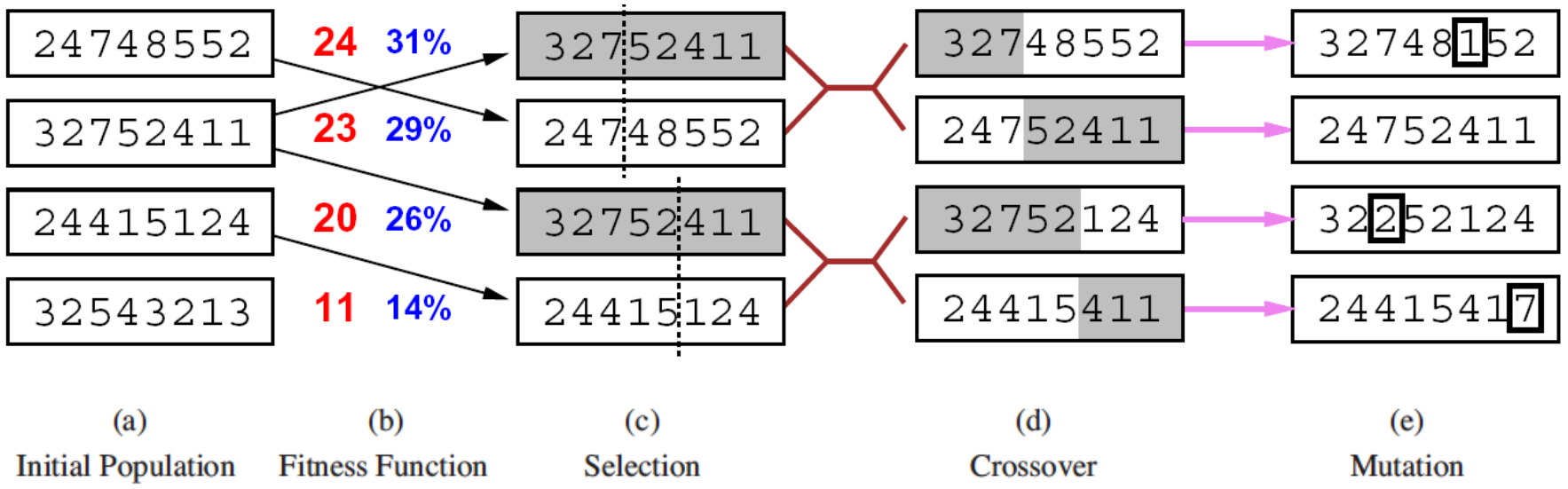


The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Genetic Algorithms: Fitness Function

- In (b), each state is rated by the objective function, or (in GA terminology) the fitness function.
  - A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of nonattacking pairs of queens, which has a value of 28 for a solution.
  - The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

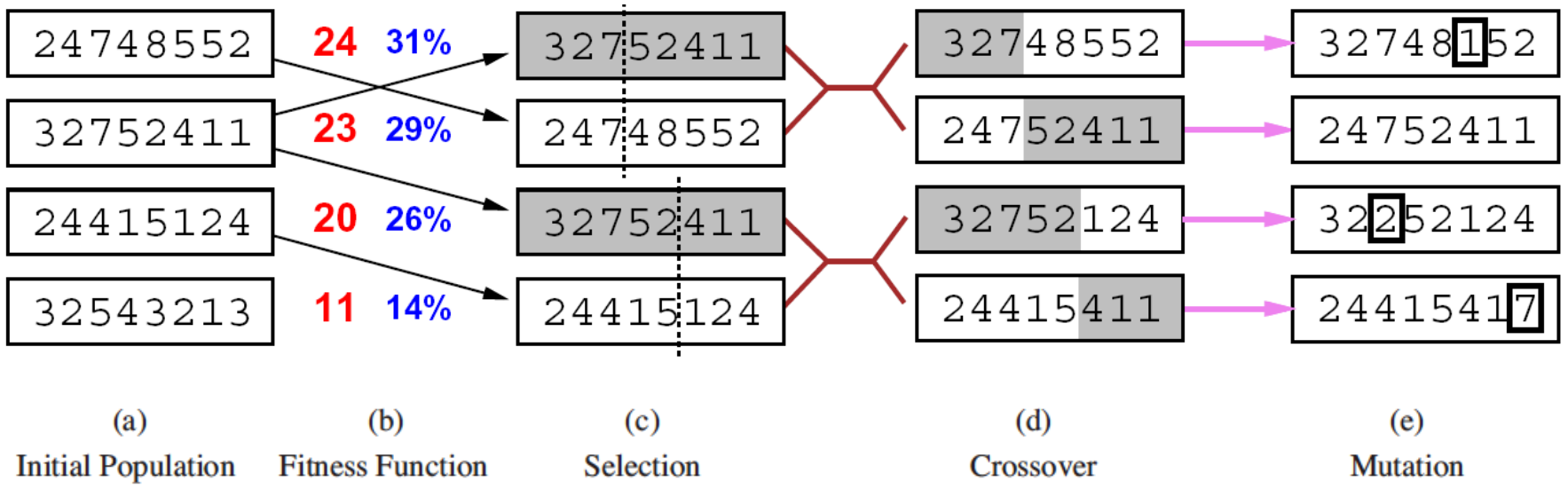
Genetic Algorithms



The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- In (c), two pairs are selected at random for reproduction, in accordance with the probabilities in (b).
  - Notice that one individual is selected twice and one not at all.
- For each pair to be mated, a crossover point is chosen randomly from the positions in the string.
  - In the example,the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.
- In (d), the offspring themselves are created by crossing over the parent strings at the crossover point

Genetic Algorithms



The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- Finally, in (e), each location is subject to random mutation with a small independent probability.
  - One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

Genetic Algorithms