L4

Wednesday, August 6, 2025 3:23 PM

L04 - Local Search an...

Local Search and Optimization Problems

Dr. Sandareka Wickramanayake

1

Outline

- Hill Climbing
- Simulated Annealing
- Local Beam Search
- Genetic Algorithms

2

Reference

- Artificial Intelligence - A Modern Approach – Chapter 4 – Section 1

3

Search Types

- Backtracking state-space search
- **Local Search and Optimization**
- Constraint satisfaction search
- Adversarial search

4

Local Search and Optimization

- Previous searches: keep paths in memory and remember alternatives so search can backtrack. The solution is a path to a goal.
- Path may be irrelevant if only the final configuration is needed
  - 8-queens
  - IC design
  - Network optimization
  - Factory floor layout
  - Job shop scheduling
  - Automatic programming
  - Cop planning etc.

5

Local Search

- Use a single current state and move only to neighbours.
  - No track of the paths
  - No track of the set of states that have been reached
- Not systematic ➔ Might never explore a portion of the search space where the solution exists.
- Advantages:
  - Use little space
  - Can find reasonable solutions in large or infinite (continuous) state spaces for which the other algorithms are unsuitable.

6

What is Local Search?

Local Search algorithms operate on a **single current state** (or a small population of states) and iteratively move to a neighboring state that is "better." They do not retain a search tree or remember paths. They have two key advantages:

1. **Very Low Memory Use:** They only need to keep track of the current state and its immediate neighbors, not the entire history of the search.
2. **Suitability for Large Problems:** They can often find good solutions in large or infinite state spaces where systematic algorithms would fail due to memory constraints.

The goal of local search is often **optimization**—finding the best possible state according to an **objective function**.

Optimization

- Local search is often suitable for optimization problems. Search for the best state by optimizing an objective function.
- F(x) where often x is a vector of continuous or discrete values.
- Begin with a complete configuration.
- A successor of state S is S with a single element changed.
- Move from the current state to a successor state.
- Low memory requirements, because the search tree or graph is not maintained in memory (paths are not saved).

7

1. The Core Idea: Optimization

- **"Local search is often suitable for optimization problems."**
  - This is the key takeaway. Optimization problems are about finding the **best** solution according to a specific measure, not just "a" valid solution.
- **"Search for the best state by optimizing an objective function."**
  - The guide for the search is not a goal test ("is this the end?") but an **objective function** (also called a cost function, loss function, or fitness function). This function assigns a value to every possible state, quantifying how "good" it is.

2. The Objective Function: `F(x)`

- **`F(x)` where often `x` is a vector of continuous or discrete values.**
  - This is a mathematical way to describe a solution or a **configuration**.
  - `x` is the **state** or **solution candidate**. It's a vector, meaning it's a list of parameters or decisions that fully define the configuration.
  - **Examples:**
    - **8-Queens:** `x` could be a vector of 8 numbers, where each number represents the row position of the queen in its column. `x = [3, 1, 4, 2, 8, 5, 7, 6]`. This is a vector of **discrete values** (integers from 1 to 8).
    - **Factory Layout:** `x` could be a list of all machine IDs, representing their assigned locations on the factory floor.
    - **Drone Design:** `x` could be a vector of **continuous values** like `[wing_span, motor_power, battery_capacity, body_weight]`. `F(x)` would then calculate the resulting flight time.

1. **"Begin with a complete configuration."**
   - Unlike pathfinding which starts from a single start state and builds a solution, local search starts with a **full, but probably bad, solution**.
   - **Example:** For 8-queens, we don't start with an empty board. We start with all 8 queens already on the board, placed randomly. It will have many conflicts, but it's a complete configuration.
2. **"A successor of state `S` is `S` with a single element changed."**
   - This defines what a "**neighbor**" is. It's a new state generated by making a **small, incremental change** to the current state.
   - **Example (8-Queens):** A neighbor is generated by picking one queen (one element of the vector `x`) and moving it to a different row in its same column. All other queens stay fixed.
3. **"Move from the current state to a successor state."**
   - The algorithm evaluates the neighbors using the objective function `F(x)` and selects one to become the new current state.
   - In simple **Hill-Climbing**, it moves to the neighbor with the best value of `F(x)` (e.g., the lowest cost or highest fitness).

Examples – 8 Queens

- Find an arrangement of 8 queens on a chess board such that no two queens are attacking each other (A queen attacks any piece in the same row, column, or diagonal.).
- Start with some arrangement of the queens, one per column.
- X[j]: row of queen in column j
- Successors of a state: move one queen
- $F(x)$: # pairs attacking each other

8

Examples – Traveling Salesman Problem

- Visit each city exactly once.
- Start with some ordering of the cities.
- State representation – order of the cities visited.
- Successor state: a change to the current ordering
- $F(x)$: length of the route

9

Comparison to Tree Search Framework

- Chapter 3: start state is typically not a complete configuration.

Chapter 4: all states are

- Chapter 3: binary goal test;

Chapter 4: no binary goal test, unless one can define one in terms of the objective function for the problem (e.g., no attacking pairs in 8-queens).

- Heuristic function: an estimate of the distance to the nearest goal

Objective function: preference/quality measure – how good is this state?

- Chapter 3: saving paths

Chapter 4: start with a complete configuration and make modifications to improve it

Visualization

- States are laid out in a landscape.
- Elevation corresponds to the objective function value.
- Move around the landscape to find the highest (or lowest) peak.
- Only keep track of the current states and immediate neighbors.

A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

Local Search Algorithms

- Strategies for choosing the state to visit next
  - Hill climbing
  - Simulated annealing
- Then, an extension to multiple current states:
  - Genetic algorithms

Hill Climbing (Greedy Local Search)

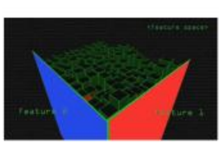
- Generate nearby successor states to the current state.
- Pick the best and replace the current state with that one.
- Loop

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*

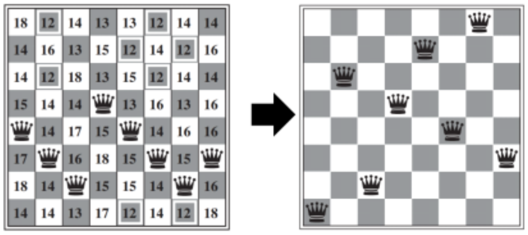
Hill Climbing (Greedy Local Search)

- Generate nearby successor states to the current state.
- Pick the best and replace the current state with that one.
- Loop



https://thumbs.gfycat.com/AlienatedImmaculateFlyingsquirrel-mobile.mp4

Hill Climbing Search Problems



15

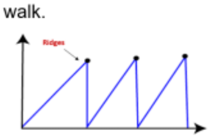
Hill Climbing Search Problems

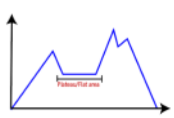
- Here, we assume maximization rather than minimization.
- Local maximum: A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.

16

Hill Climbing Search Problems

- Ridges: It is a region that is higher than its neighbors but itself has a slope.
  - It is a special kind of local maximum.
  - Results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- Plateau: the evaluation function is flat, resulting in a random walk.





17

Variants of Hill Climbing

- **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.

18

1. Stochastic Hill Climbing

Core Idea: Don't always take the *absolute best* move. Instead, choose **randomly from all moves that improve the current state**, with a bias towards better moves.

- **How it works:** Instead of evaluating every single *neighbor* and picking the very best one (steepest ascent), Stochastic Hill Climbing:
  1. Identifies all neighbors that are an improvement (all "uphill moves").
  2. Assigns each of these good moves a probability. **The steeper the uphill move (i.e., the bigger the improvement), the higher its probability of being chosen.**
  3. Randomly selects one move based on these probabilities.

2. First-Choice Hill Climbing

Core Idea: A specific, efficient way to *implement* Stochastic Hill Climbing when there are too many successors to evaluate fully.

- **How it works:** This strategy is for when a state has a massive number of neighbors (e.g., thousands or millions). Evaluating all of them to find the best one (steepest ascent) or even to build a probability distribution (stochastic) would be too computationally expensive.
  1. Instead of generating all successors, it generates them **one at a time, randomly.**
  2. The moment it finds a successor that is better than the current state, it immediately moves to it.
  3. It doesn't bother to check if there's an even *better* move available.

Variants of Hill Climbing

- **Random-restart hill climbing**
  - Start different hill-climbing searches from random starting positions stopping when a goal is found.
  - Save the best result from any search so far.
  - If all states have an equal probability of being generated, it is complete with a probability approaching 1 (a goal state will eventually be generated).
  - Finding an optimal solution becomes the question of a sufficient number of restarts.
  - Surprisingly effective, if there aren't too many local maxima or plateau.

19

Random-Restart Hill Climbing is a simple but powerful meta-algorithm:

1. **Run** a standard hill-climbing search (e.g., steepest ascent) from a **randomly generated initial state.**
2. **Stop** the search when it converges to a solution (which could be a local maximum or the global maximum).
3. **Save** the result.
4. **Repeat** steps 1-3 many times, each time from a new, random starting point.
5. **Return** the **best solution** found across all restarts.

Simulated Annealing

- A hill-climbing algorithm that never makes "downhill" moves is always vulnerable to getting stuck in a local maximum.
- A purely random walk that moves to a successor state without concern for the value will eventually discover the global maximum but will be extremely inefficient.
- Combine the ideas: add some randomness to hill-climbing to allow the possibility of escape from a local optimum.

- Simulated annealing wanders around during the early parts of the search, hopefully toward a good general region of the state space
- Toward the end, the algorithm does a more focused search, making a few bad moves.

How Simulated Annealing Works (The Smart Hiker)

Simulated Annealing is the smart hiker who combines both ideas:

1. **Early On (Exploring):** At the start, the hiker is willing to take **big risks**. They might even take a few steps **downhill**. This allows them to get off a small hill and explore different areas of the mountain range to find the best region.
2. **Later On (Exploiting):** As time passes, the hiker becomes **more cautious**. They take fewer and fewer downhill steps. By the end, they are only walking uphill, fine-tuning their position to reach the very top of the highest peak they found.

Simulated Annealing

- Based on a metallurgical metaphor.
- Annealing: harden metals and glass by heating them to a high temperature and then gradually cooling them.
- Start with a temperature set very high and slowly reduce it.

- Run hill-climbing with the twist that you can occasionally replace the current state with a worse state based on the current temperature and how much worse the new state is.
- At the start, make lots of moves and then gradually slow down.

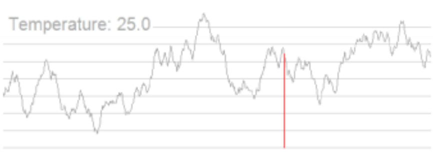
The "Annealing" Part

The name comes from **metallurgy** (working with metals). Annealing is a process where metal is heated and then slowly cooled. This slow cooling allows the atoms to settle into a strong, stable, low-energy structure (the **global minimum**).

- **Heating (Early Search):** High "temperature" means high energy and randomness, allowing for exploration.
- **Cooling (Late Search):** The "temperature" is lowered, reducing randomness and allowing the solution to settle into an optimum.

Simulated Annealing

- Generate a random new neighbor from the current state.
- If it's better take it.
- If it's worse, then take it with some probability proportional to the temperature and the delta between the new and old states.



https://upload.wikimedia.org/wikipedia/commons/d/d5/Hill_Climbing_with_Simulated_Annealing.gif

The algorithm decides what to do next by following these steps:

1. **Generate a Random Move:** Look at a random state near your current one (a "neighbor").
2. **Is it Better?** If this new state is an improvement, **always take it**. (This is the hill-climbing part).
3. **Is it Worse?** This is the special part. If the new state is worse, you don't automatically reject it. Instead, you might take it anyway with a certain **probability**.

The "Probability" and "Temperature"

The chance of making a "bad" move depends on two things:

1. **How Much Worse?** A slightly worse move is more acceptable than a disastrously worse one.
2. **The Temperature:** This is the most important part.

- **High Temperature (Early in the search):** The system is "hot." This makes the probability of taking a bad move **high**. The algorithm is adventurous and explores wildly.
- **Low Temperature (Later in the search):** The system has "cooled down." The probability of taking a bad move becomes **very low**. The algorithm becomes cautious and fine-tunes its solution.

Your Specific Example: `Temperature: 25.0`

The number **25.0** is the current temperature. To understand what it means, we need to know:

- **Is 25.0 high or low?** We need to know the starting temperature (e.g., 1000) and the ending temperature (e.g., 0.1).
- **If the starting temperature was 100,** then 25.0 is fairly cool. The algorithm is likely in a later phase, becoming more cautious but still allowing for some occasional "bad" moves to escape very small local optima.
- **If the starting temperature was 30,** then 25.0 is still very hot, and the algorithm is still in its highly exploratory, random phase.

In short: **The temperature controls the algorithm's willingness to take risks. A value of 25.0 means it is currently willing to take some risks, but we don't know how many without more context.**

Simulated Annealing

```
function Simulated-Annealing(start, schedule)
    current ← start
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T=0 then return current
        next ← a randomly selected successor of the current
        ΔE ← Value[next] − Value[current]
        if  ΔE > 0 then current ← next
        else current  ← next only with probability e^(ΔE/T)
```


The Simulated Annealing Recipe

Imagine you're trying to find the highest point in a foggy landscape. Here's your instruction manual:

1. **Start:** Begin where you are (`current ← start`).
2. **Check the Time & Temperature:** Look at your schedule. The `schedule` is a timer that tells you how "hot" the system is at each step `t`. As time `t` increases, the temperature `T` decreases.
3. **Stopping Condition:** If the temperature `T` has cooled down to zero, stop. You're done. Return your current location (`return current`).
4. **Make a Random Move:** Pick a new location completely at random from the ones right next to you (`next ← a randomly selected successor`).
5. **Check if it's Better:** Calculate how much higher this new location is compared to your old one (`ΔE ← Value[next] − Value[current]`).
    - **If it's higher (ΔE > 0):** Great! Move there. (`current ← next`). This is like walking uphill.
6. **Check if it's Worse (The Magic Step):** If the new location is lower (ΔE is negative), you don't automatically stay put. You might still move there!
    - **Calculate the Probability:** The chance you move downhill is given by the formula: probability = $e^(\Delta E / T)$
    - **How it works:**
        - `ΔE` **is negative** (because it's a worse state), so `ΔE / T` is a negative number.
        - A large, negative exponent makes the probability (`e^(negative number)`) a small number between 0 and 1.
        - **The Role of `T` (Temperature):**
            - **High `T` (Hot):** `ΔE / T` is a small negative number. `e^(small negative)` is a probability close to 1. You are **very likely** to take the bad step. (Lots of exploration).
            - **Low `T` (Cool):** `ΔE / T` is a large negative number. `e^(large negative)` is a probability close to 0. You are **very unlikely** to take the bad step. (Fine-tuning).

Simulated Annealing

- Probability of a move decreases with the amount ΔE by which the evaluation is worsened.
- A second parameter T is also used to determine the probability: high T allows worse moves, T close to zero results in few or no bad moves.
- Schedule input determines the value of T as a function of the completed cycles.

Even after finding the global maximum, the algorithm will almost certainly leave it and explore other local maxima.

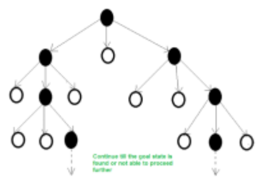
The algorithm starts **hot and reckless**, freely moving downhill to explore the entire landscape. As the "temperature" **cools down** according to the schedule, it becomes **increasingly conservative**, acting more like a standard hill-climber that only moves uphill, locking onto the best peak it found during its exploratory phase.

The genius is in using that probability formula `e^(ΔE / T)` to mathematically manage the transition from **explorer** to **exploiter**.

Local Beam Search

something in between BFS and best first search

- Keep track of k states rather than just one, as in hill climbing
- Begins with k randomly generated states
- At each step, all successors of all k states are generated
- If anyone is a goal, algorithm halts
- Otherwise, selects the best k successors from the complete list, and repeats



25

Local Beam Search Explained Simply

Imagine you have a team of k hikers (e.g., k=5) instead of just one.

1. **Start:** The team fans out. Each hiker starts on a random hill in the mountain range.
2. **Look Around:** Each hiker looks at all the points immediately around them (their "successors").
3. **Radio In:** All hikers radio back to base camp the quality (height) of *all* the points they can see.
4. **New Team Selection:** Base camp looks at this combined list of *all possible next steps from every hiker*. It then picks the k very best points from this giant list.
5. **Move:** The team of hikers all moves to these new k best points. (Note: They might all converge onto the same hill, or they might spread out onto different ones).
6. **Repeat:** They repeat the process from these new points until one of them finds the summit (the goal).

Local Beam Search

- Successors can become concentrated in a small part of state space
- Stochastic beam search: choose k successors with the probability proportional to the successor's value.
  - Increase diversity

26

The Problem: Lack of Diversity

The standard Local Beam Search has a big flaw: **The entire search can quickly get stuck.**

Imagine our team of k hikers:

- If most of them are on the same big mountain, the "best" next steps will all be on that same mountain.
- Soon, all k hikers will be clustered on that one mountain, completely ignoring other potentially taller mountains in the range.
- The search loses **diversity** and behaves like a single, slightly more powerful hill-climber, vulnerable to local maxima.

The Solution: Stochastic Beam Search

To fix this, we introduce randomness, similar to Simulated Annealing. Instead of always choosing the absolute k best successors, we choose them **probabilistically**.

How it works (The Lottery Analogy):

1. **Generate all successors** from all k current states. Assign each successor a "ticket" for a lottery.
2. **The "value" of a successor determines how many tickets it gets.** A higher-valued (better) state gets more tickets. A lower-valued state gets fewer, but it still gets some.
3. **Randomly draw k tickets** from the pile. The successors whose tickets are drawn get to be the next generation of states.

Genetic Algorithms

- Local beam search, but…
  - A successor state is generated by ***combining two parent states***.
- Start with k randomly generated states (population).
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s).
- Evaluation function (fitness function). Higher = better.
- Produce the next generation of states by selection, crossover, and mutation.

27

Imagine you're trying to breed the fastest racehorse. You don't know the perfect DNA sequence, but you can have a group of horses and race them.

1. **Population:** You start with a diverse group (k random horses). This is your initial **population.**
2. **Fitness:** You race them. The finish time is your **fitness function**. Faster horses are "fitter".
3. **Selection:** You pick the best horses to be **parents** for the next generation. Fitter horses are more likely to be chosen.
4. **Crossover (The Key Idea):** You create a foal (a **successor**) by combining the DNA of two parent horses. For example, the front half of its DNA comes from one parent, the back half from the other. This is how you "combine two parent states".
5. **Mutation:** You randomly tweak a small part of the foal's DNA (e.g., change one gene). This introduces new traits that might not exist in the current population, preventing the search from getting stuck.
6. **New Generation:** The new generation of foals replaces the old generation. You repeat the process: race them, select the best, breed them, etc.

The Algorithm Steps (Summary)

1. **Initialization:** Generate a random population of k individuals (strings of DNA).
2. **Fitness Evaluation:** Test each individual and assign it a fitness score.
3. **Selection:** Choose pairs of parents for mating, with a probability based on their fitness.
4. **Crossover:** For each pair, create one or two offspring by swapping segments of the parents' strings.
5. **Mutation:** Randomly flip some bits (or change some characters) in the offspring with a small probability.
6. **Replacement:** Form a new generation from the offspring (and sometimes the best parents).
7. **Termination:** Repeat steps 2-6 until a satisfactory solution is found or a set number of generations have passed.

Genetic Algorithms

Specific Mechanics

- Representation of individuals
  - Classic approach: an individual is a string over a finite alphabet, with each element in the string called a gene
  - Usually, binary instead of AGTC as in real DNA
- Mixing number
- Selection strategy
  - Random
  - Selection probability proportional to fitness
  - Selection is made with replacement to make a very fit individual reproduce several times

28

1. Representation of Individuals (The "DNA")

This is how you design a solution so that a computer can "evolve" it.

- **The Concept:** In nature, an individual's blueprint is its DNA, a string of genes (A, T, C, G). In a GA, an individual's blueprint is a string of data that represents a possible solution to your problem.
- **"Classic approach":** The most common way is to represent this blueprint as a **string of 0s and 1s** (a binary string). This is like a simplified version of real DNA.
  - **Example:** If you're trying to find the maximum of a function, the binary string might represent a number in binary form (e.g., 1011 represents the number 11).
  - **Gene:** Each single element (each 0 or 1) in the string is called a **gene**. It's the smallest unit of the solution.

Why binary? It's simple, universal, and makes the next steps (crossover and mutation) very easy to program.

2. Mixing Number

This is a more technical term related to how parents create offspring.

- **The Concept:** In nature, two parents mix their DNA to create a child. The "mixing number" in GAs specifies **how many parents are involved** in creating a single offspring.
- **The Standard:** The mixing number is almost always 2. This is called **sexual crossover**. Two parents are selected, and their strings are spliced together to create one or two children.
- **Other possibilities:** While rare, some schemes might use 1 parent (asexual reproduction, essentially just mutation) or even more than 2 parents to create a single offspring.

3. Selection Strategy (The "Mating Pool")

This is the rule for deciding which individuals in the current generation get to be parents and pass on their genes.

- **Random:** Choose parents completely randomly from the population. This is a bad strategy because it ignores fitness and doesn't drive improvement.
- **Fitness-Proportional Selection (The Common Method):** This is the strategy described in your image.
  - **How it works:** The chance of an individual being selected as a parent is **directly proportional to its fitness score**. The fitter it is, the higher its chance of being chosen.
  - **Analogy:** Imagine a lottery where each individual gets tickets. A very fit individual gets 100 tickets. A less fit individual gets only 5 tickets. You then draw randomly from all the tickets. The individual with 100 tickets has a much higher chance of winning (being selected).
  - **"Selection with replacement":** This is a crucial detail. It means the same individual can be selected **more than once** to be a parent. That very fit individual with 100 tickets might win the lottery multiple times! This allows the best solutions to produce many offspring, quickly spreading their good "genes" through the population.

Genetic Algorithms

- Recombination procedure
  - Random pairing of selected individuals
  - Random selection of cross-over points
- Mutation rate – How often offspring have random mutations to their representation.
- The makeup of next generation
  - Forms with the newly generated offspring
  - Elitism - Include a few top-scoring parents from the previous generation in addition to newly formed offspring.
  - Culling – Discard individuals below a given threshold.

1. **Recombine** selected parents to create new offspring.
2. **Mutate** those offspring to add new variety.
3. **Build** the next generation, often using **elitism** to protect the best solution found so far.

29

Imagine you have your selected parents. Now you need to create their children. This is the **recombination procedure**:

1. **Random Pairing:** You randomly shuffle the list of chosen parents and put them into pairs. A super-fit parent might be paired with another super-fit one, or with a less-fit one. This randomness is good for diversity.
2. **Random Crossover Point:** For each pair, you create children by swapping parts of their "DNA" (their string). You choose a random spot to cut and swap.
   - **Example:** Parent A: AAAA, Parent B: BBBB. Crossover point after the 2nd gene.
   - Child 1: AA + BB = AABB
   - Child 2: BB + AA = BBAA

This is how good traits from different parents get mixed together.

Adding Spice: Mutation Rate

- **What it is:** After a child is created, you go through its new string and give each gene a small chance to randomly change.
- **Why:** Crossover can only mix existing traits. **Mutation creates brand new traits** that might lead to a much better solution. It prevents the population from becoming too similar and getting stuck.
- **The Rate:** This is a small probability (e.g., a 0.1% chance per gene). Too low, and you get no innovation. Too high, and the search becomes random and chaotic.

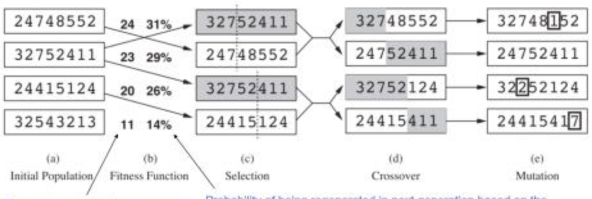
Building the New Generation

Once you have a pool of new offspring, you have to decide who forms the next generation. There are a few ways to do this:

- **Standard Way:** The next generation is formed **only from the newly generated offspring**. The parents are completely replaced.
- **Elitism (Smart Upgrade):** This is a very common and effective trick.
  - You first ** forcefully save the absolute best parent(s)** from the previous generation.
  - You then add them to the new generation of offspring.
  - **Why it's great:** It guarantees that your best solution ever found is never lost. You only ever get better or stay the same; you never accidentally lose your champion.
- **Culling (The Harsh Method):**
  - You discard any individual—whether parent or offspring—that has a fitness score below a certain level.
  - This aggressively pushes the population toward higher fitness but can be risky if it reduces diversity too much.

Example – N-Queens

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



fitness: #non-attacking queens

Probability of being regenerated in next generation based on the fitness value.

(a) Initial Population

The algorithm starts with 4 random individuals (potential solutions):

1. 24748552
2. 32752411
3. 24415124
4. 32543213

(b) Fitness Function

The fitness of a solution is the number of **non-attacking pairs of queens**. Since there are 8 queens, the maximum number of non-attacking pairs is the goal. Higher fitness is better.

The image calculates this fitness for each individual:

- 24748552 has a fitness of 24
- 32752411 has a fitness of 23
- 24415124 has a fitness of 20
- 32543213 has a fitness of 11

(c) Selection

The algorithm creates a "mating pool" for the next generation. The probability of an individual being selected as a parent is proportional to its fitness.

The image shows this probability:

- 24748552 : Fitness 24 -> 31% chance
- 32752411 : Fitness 23 -> 29% chance
- 24415124 : Fitness 20 -> 26% chance
- 32543213 : Fitness 11 -> 14% chance

The fittest individual (24748552) is most likely to be chosen to pass on its "genes".

(d) Crossover

Selected parents are paired up and "bred". A random crossover point is chosen, and parts of their strings are swapped to create children.

The image shows this happening:

- The first parent (24748552) and the second parent (32752411) are crossed over.
- The result is two new offspring: 32748552 and 24752411 .

You can see how segments of the parent strings were combined.

(e) Mutation

After crossover, genes (digits) are ra

The image shows a mutation happen

- One of the offspring, 24415411 , h
24415417 .

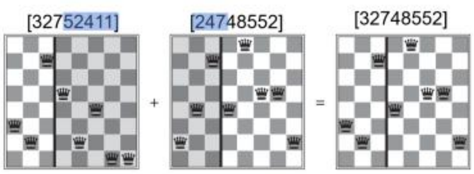
The Result

The process of selection, crossover, an
24752411 , 32752124 , 24415417).

This new generation will, on average,
parts of the previous generation. This
valid solution to the 8-queens proble

The final string [16257483] is likely a
find.

Example – N-Queens



- Has the effect of "jumping" to a completely different new part of the search space (quite non-local)

Genetic Algorithms

```
function GA (pop, fitness-fn)
  Repeat
    new-pop = {}
    for i from 1 to size(pop):
      x = rand-sel(pop,fitness-fn)
      y = rand-sel(pop,fitness-fn)
      child = REPRODUCE(x,y)
      if (small rand prob): child ←
   mutate(child)
      add child to new-pop
    pop = new-pop
  until an indiv is fit enough, or out of
   time
  return best indiv in pop, according to
   fitness-fn
```

```
function REPRODUCE(x,y)
  n = len(x)
  c = random num from 1
to n
  return  APPEND
(substr(x,1,c),substr(y,
c+1,n)
```

Comments on Genetic Algorithms

- Genetic Algorithm is a variant of "stochastic beam search".
- Positive points
  - Random exploration can find solutions that local search can't (via crossover primarily).
  - Appealing connection to human evolution.
- Negative points
  - Many "tunable" parameters
  - Difficult to replicate performance from one problem to another
  - Lack of good empirical studies comparing to simpler methods
  - Useful on some (small?) problems, but no convincing evidence that GAs are generally better than hill-climbing w/random restarts.

The first point is a great summary: a GA is essentially a **Stochastic Beam Search** where the way you generate new states (successors) is not by looking at neighbors, but by **breeding** existing states together through crossover and mutation. It's a parallel, population-based search.

The Positive Points (The Hype)

1. **Powerful Exploration via Crossover:** This is the main argument for GAs. Crossover isn't just a random change; it's a way of combining good "building blocks" from different parents. This can lead to intelligent jumps in the search space that simple hill-climbing or random restarts might miss, potentially finding more complex solutions.
2. **Biologically Inspired Appeal:** The idea of solving problems by simulating evolution is intellectually fascinating and intuitively appealing. It feels powerful and natural.

The Negative Points (The Reality Check)

1. **Many "Tunable" Parameters:** This is a huge practical challenge. To make a GA work, you have to choose:
   - Population size
   - Mutation rate
   - Crossover rate
   - Selection strategy
   - Elitism count
   - ...and more!

   Getting these wrong can lead to terrible performance. This makes GAs feel less like a ready-to-use tool and more like a complex machine that needs careful tuning.
2. **Performance Doesn't Always Transfer:** A set of parameters that works miracles on one problem might fail completely on a slightly different problem. You often can't just take a GA from one application and expect it to work on another without starting the tuning process over.
3. **Lack of Conclusive Evidence:** This is a major criticism from computer scientists. For many problems, a much simpler algorithm like **Hill-Climbing with Random Restarts** (just start from a random point, climb until stuck, then start over from a new random point) is just as effective, and often faster and easier to implement, than a complex GA. There isn't overwhelming proof that GAs are *generally* superior.

- GAs are **useful and powerful** for certain types of problems, especially those where solutions can be built from good sub-parts (building blocks).
- However, there is **no "magic bullet"**. For many tasks, simpler, less glamorous algorithms are just as good or better.
- They are a valuable tool in the algorithm toolbox, but not a default winner for every optimization problem.

ndomly changed with a small probability. This introduces new traits.

ing:

as its last gene mutated from a `1` to a `7`, creating a new individual:

d mutation creates a **new generation** of individuals (e.g., `32748552`,

be fitter than the previous one because it was built from the best
cycle repeats until it finds an individual with a perfect fitness score (a
n).

n example of such a solution that the algorithm could eventually