CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Metaheuristic Optimization

Recap

Dr. Diarmuid Grimes

Module

- **Complexity of combinatorial Problems**
- Population-based
  - Genetic Algorithms
  - Ant Colony Optimization
- Single-solution
  - Local Search Algorithms

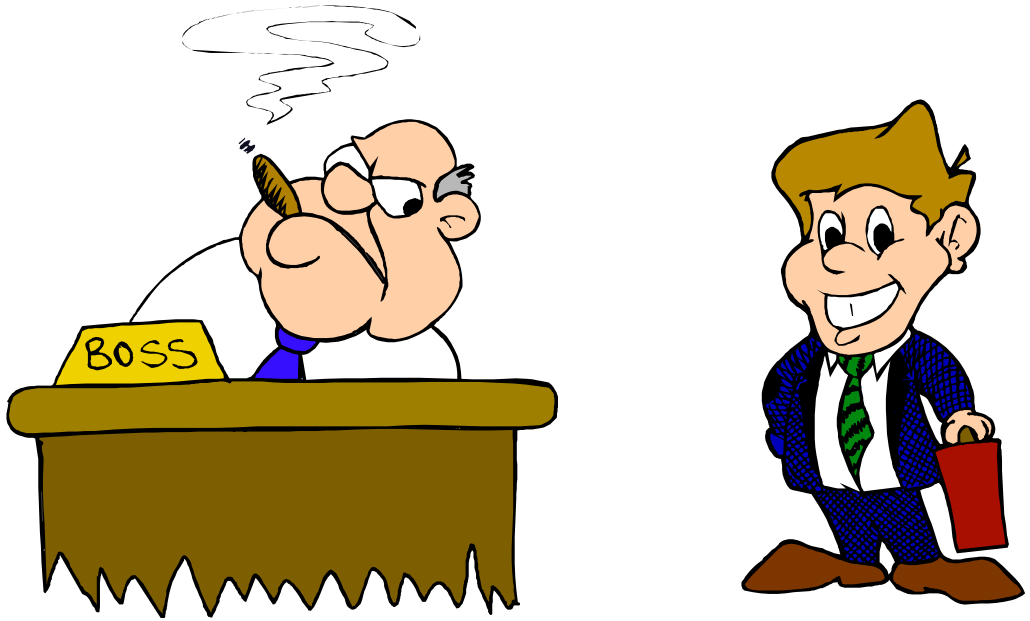
Dealing with Hard Problems

What to do when we find a problem that looks hard…

I couldn't find a polynomial-time algorithm;
I guess I'm too dumb.

Dealing with Hard Problems

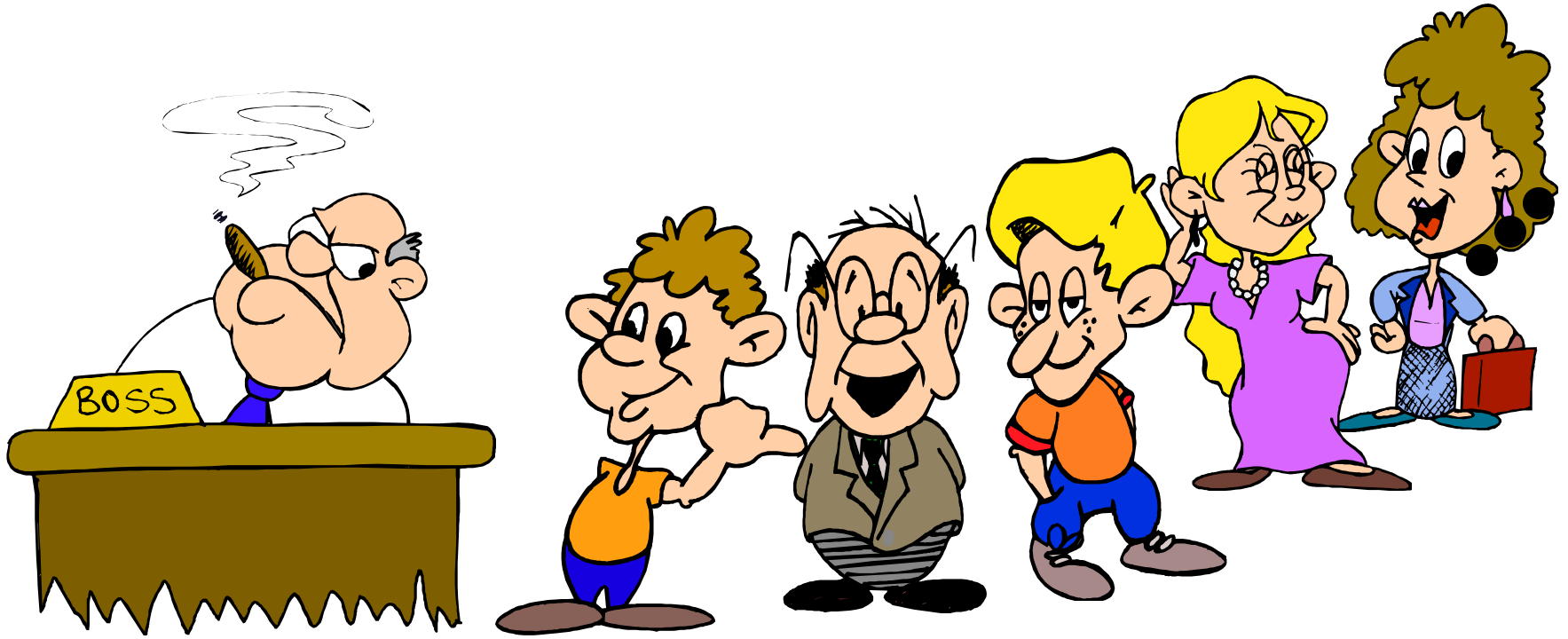
Sometimes we can prove a strong lower bound… (but not usually)



I couldn't find a polynomial-time algorithm,
because no such algorithm exists!

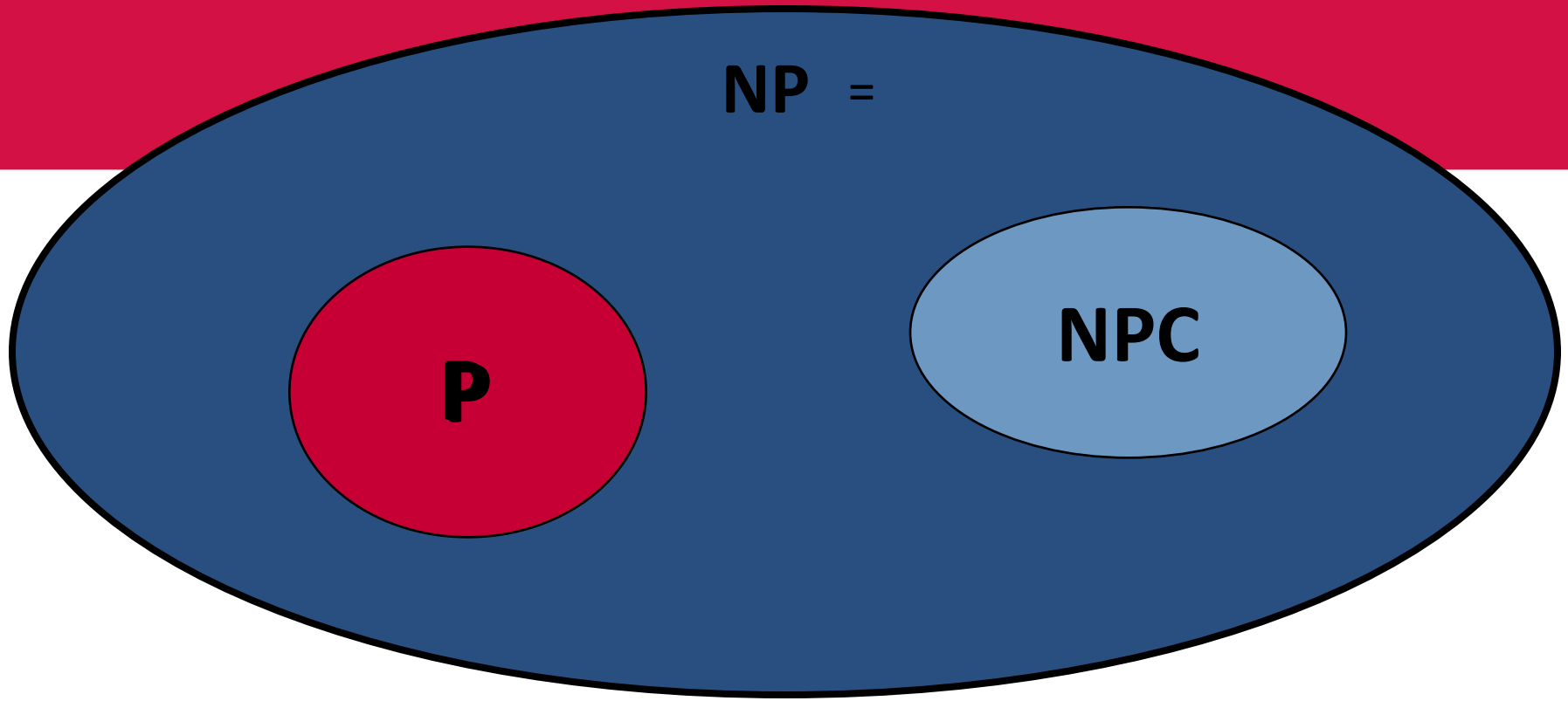
Dealing with Hard Problems

NP-completeness let's us show collectively that a problem is hard.



I couldn't find a polynomial-time algorithm,
but neither could all these other smart people.

The Theory of NP-Completeness



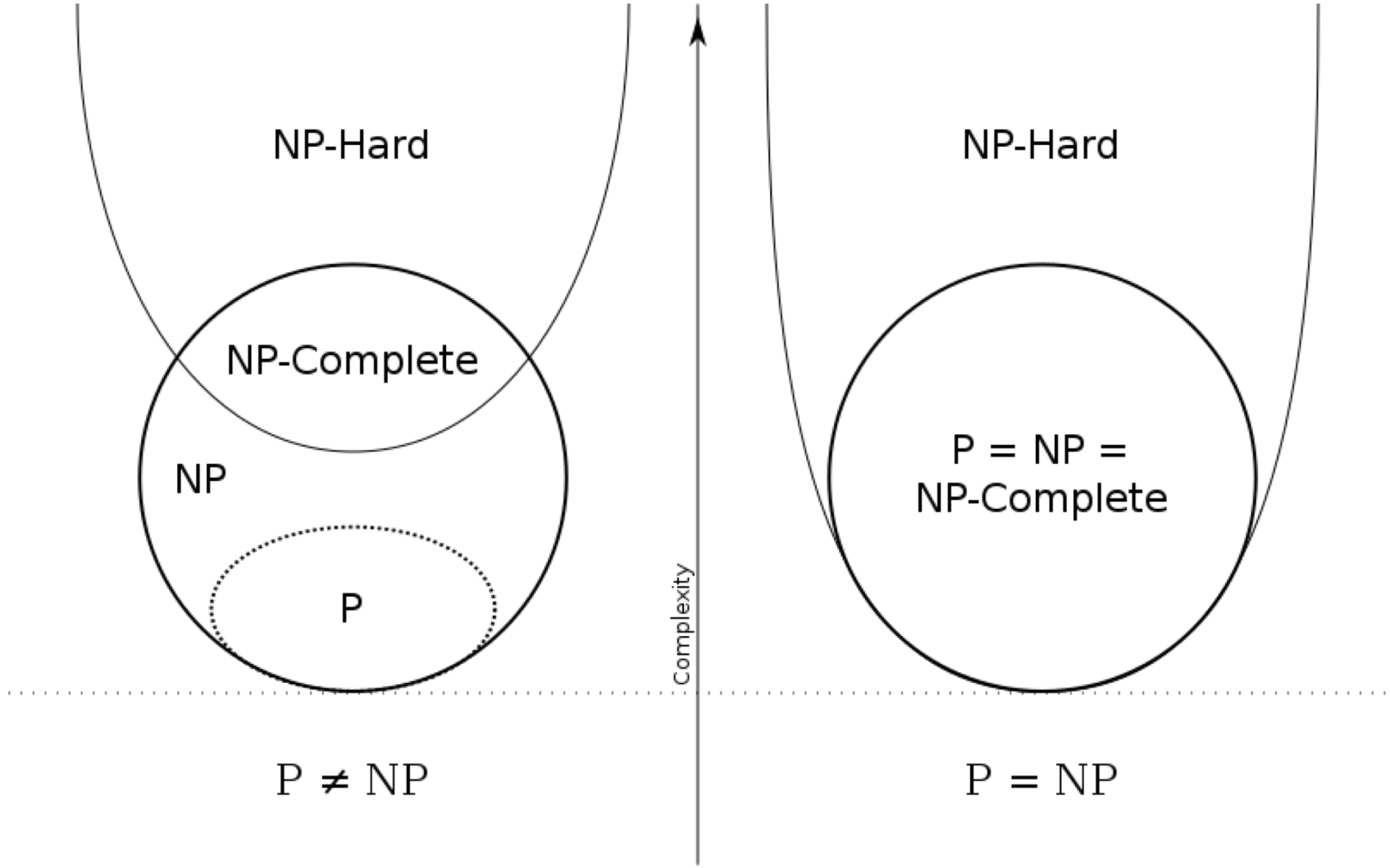
P =? NP

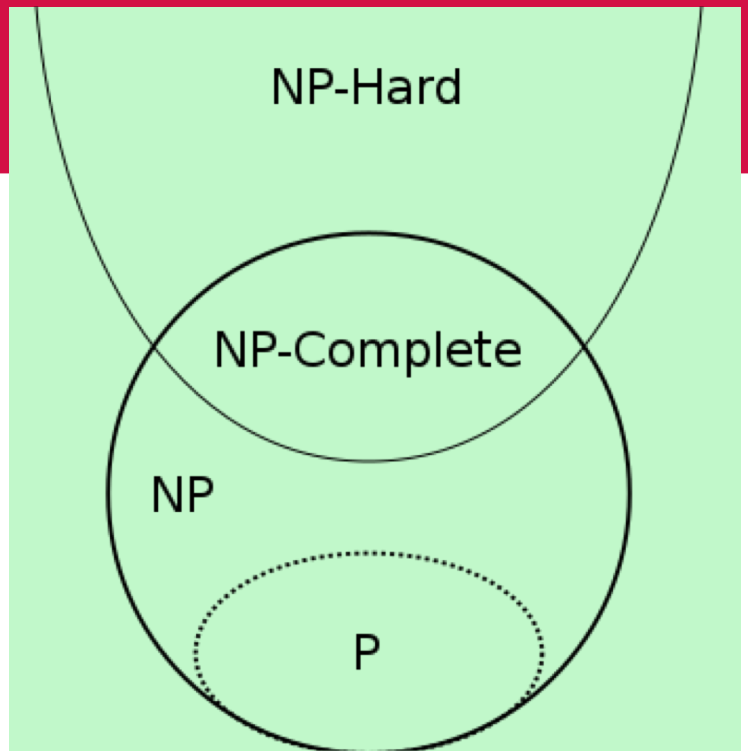
NP: Non-deterministic Polynomial

P: Polynomial

NPC: Non-deterministic Polynomial Complete

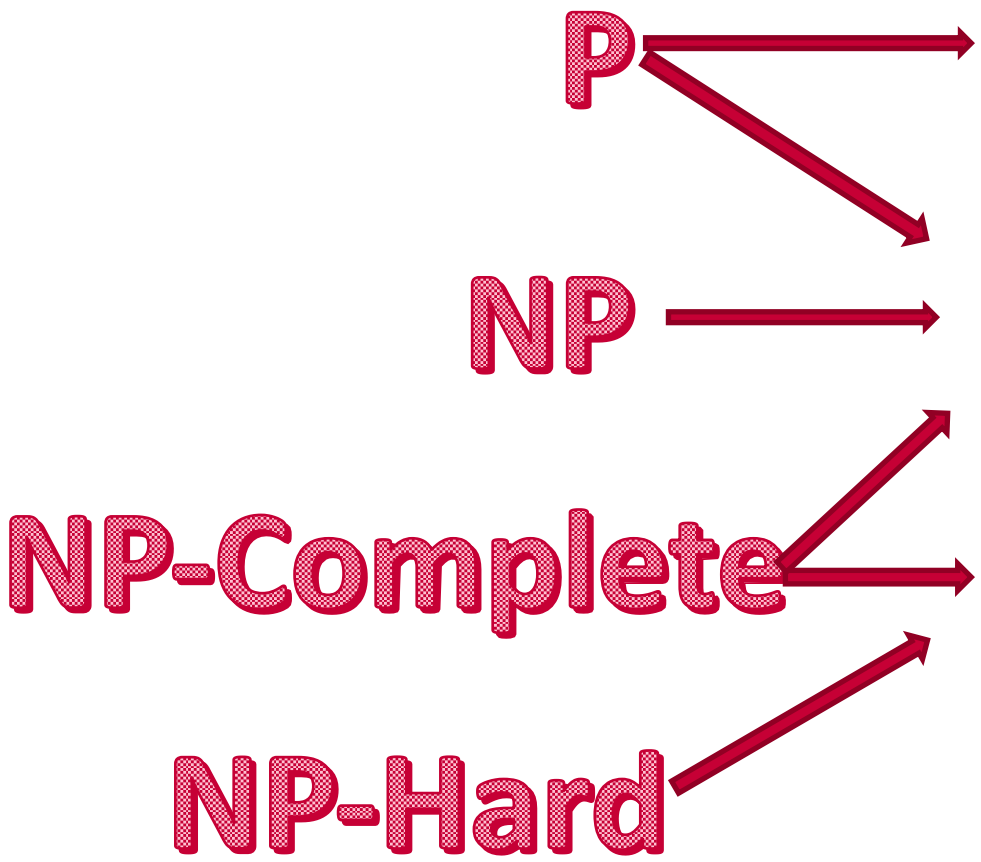
Recap


NP-Hard
NP-Complete
NP
P
Complexity
NP-Hard
P = NP =
NP-Complete
P ≠ NP
P = NP



- **P**: the class of problems which can be solved by a deterministic polynomial algorithm.
- **NP** : the class of decision problem which can be solved by a non-deterministic polynomial algorithm.
- **NP-hard**: the class of problems to which every NP problem reduces.
- **NP-complete (NPC)**: the class of problems which are NP-hard and belong to NP.

Solving and Verifying Solutions



1. Decision problem can be solved in polynomial time
2. Candidate solution for problem can be verified in polynomial time
3. All problems in NP can be *reduced* in polynomial time to this problem

Relationship Between NP and P

- It is not known whether **P**=**NP** or whether **P** is a proper subset of **NP**
- It is believed **NP** is much larger than **P**
  - But no problem in **NP** has been proved as not in **P**
  - No known deterministic algorithms that are polynomially bounded for many problems in **NP**
  - So, “does **P** = **NP**?” is still an open question!

Definition of NP-Complete

L is NP-complete if and only if

(1) L is in NP and

(2) for all L' in NP, $L' \leq_p L$.

In other words, L is at least as hard as every language in NP.

Implication of NP-Completeness

Theorem: Suppose L is NP-complete.

(a) If there is a poly time algorithm for L, then P = NP.

(b) If there is no poly time algorithm for L, then there is no poly time algorithm for any NP-complete language.

Showing NP-Completeness with a Reduction

To show L is NP-complete:

(1) Show L is in NP.

(2.a) Choose an appropriate known NP-complete language L'.

(2.b) Show $L' \leq_p L$.

Why does this work? By transitivity: Since every language L'' in NP is polynomially reducible to L', L'' is also polynomially reducible to L.

Showing 3SAT is NP-Complete by Reduction

(1) To show 3SAT is in NP, use same algorithm as for SAT to verify a candidate solution (truth assignment)

(2.a) Choose SAT as known NP-complete problem.

(2.b) Describe a reduction from SAT inputs to 3SAT inputs

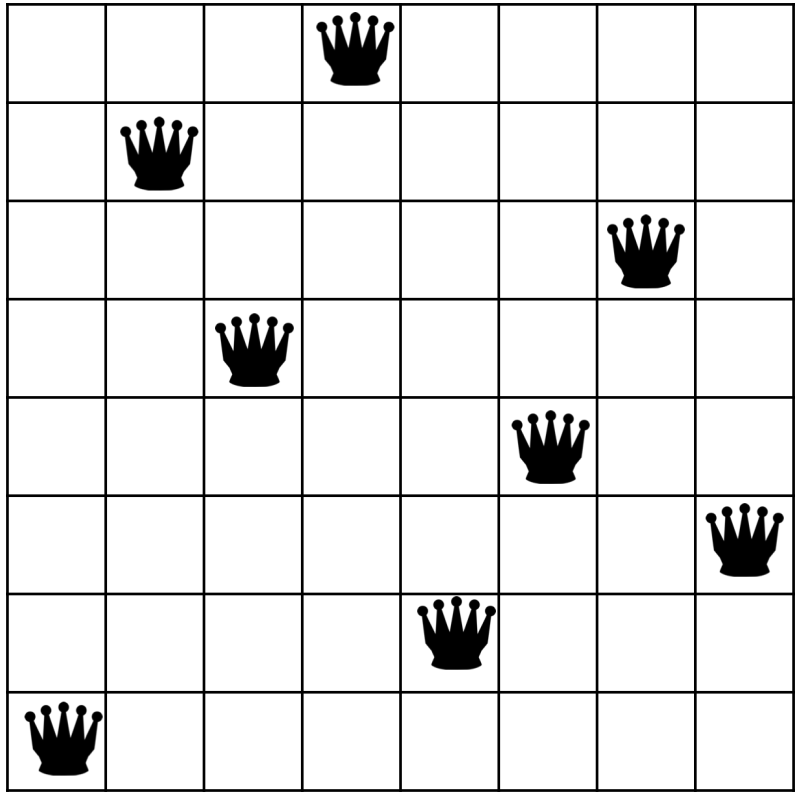
- computable in poly time
- SAT input is satisfiable iff constructed 3SAT input is satisfiable

- Every instance of SAT can be represented as 3SAT such that
  - if the SAT instance is unsatisfiable then the 3SAT instance is unsatisfiable,
  - Else, every solution for the SAT instance must also be a solution for the 3SAT instance

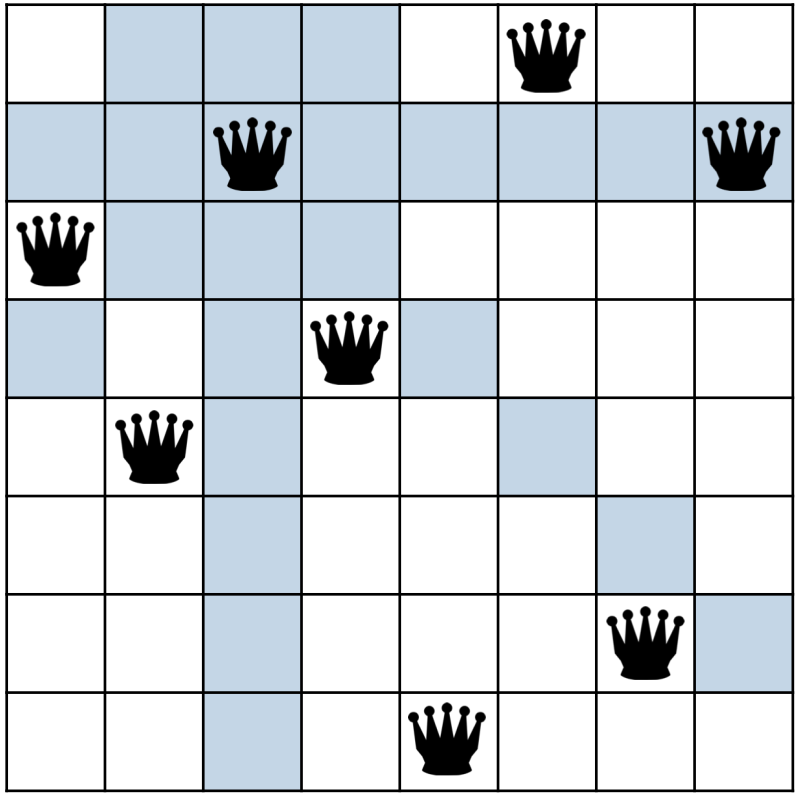
N-Queens Problem -> Satisfaction Problem

Task: place 8 queens on the chess board such that they do not attack each other

Good



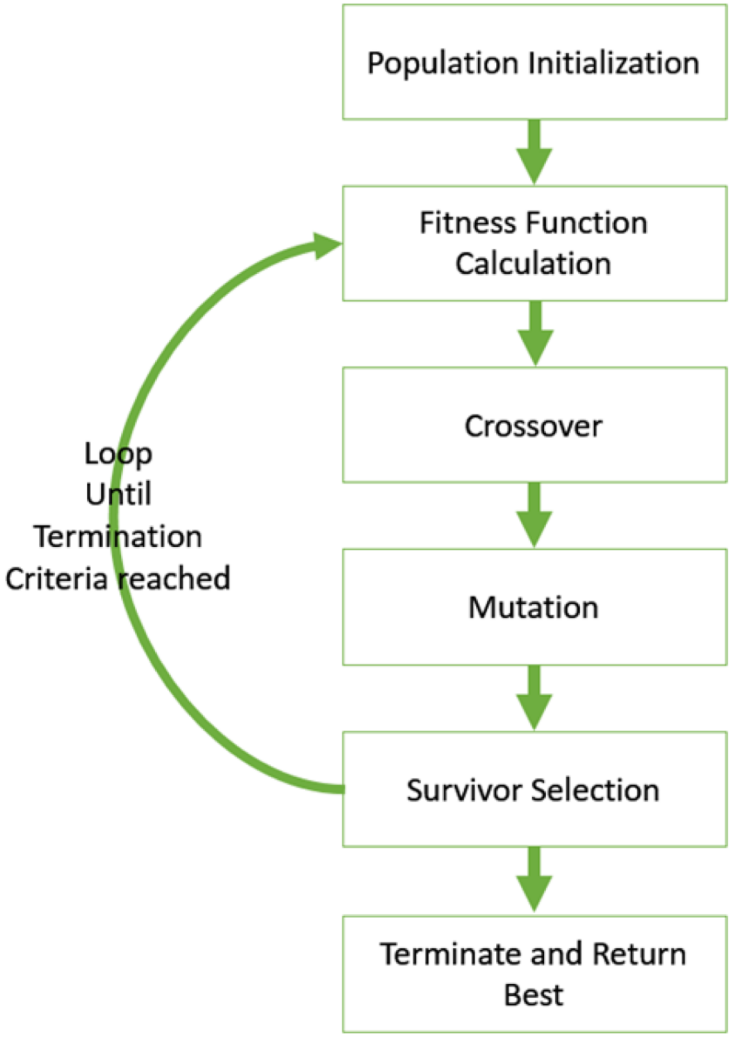
Bad

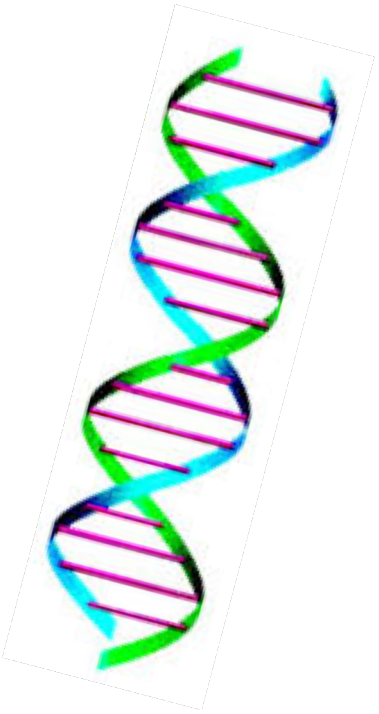


Classical search vs. Local Search

Classical search	Local Search
➢ systematic exploration of search space. ➢ Keeps one or more paths in memory. ➢ Records which alternatives have been explored at each point along the path. ➢ The **path** to the goal is a **solution** to the problem.	➢ In many optimization problems, the path to the goal is **irrelevant; the goal state itself is the solution**. ➢ State space = set of "complete" configurations. ➢ Find configuration satisfying constraints, Find best state according to some objective function h(s). e.g., **n-queens**, h(s)= number of attacking queens. **In such cases, we can use Local Search Algorithms.**

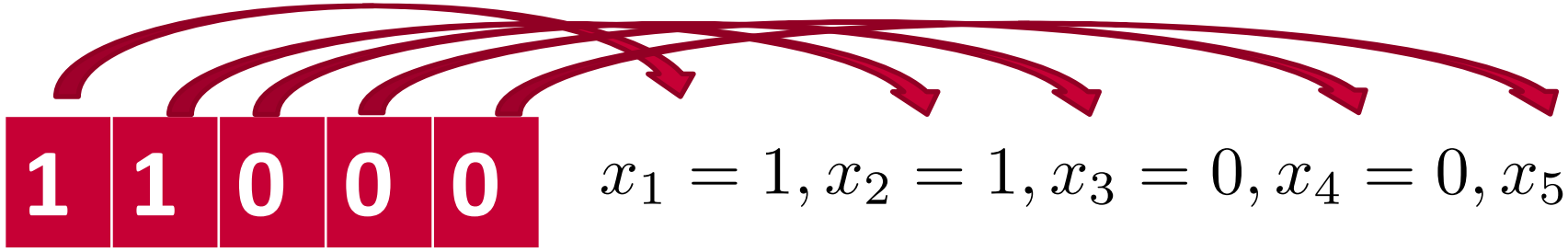
Evolutionary computation





Terminology

$$(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_3) \wedge (x_5)$$



$$x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 0$$

Number of Unsatisfied clauses

Cost -> 2

$$x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 1$$

Cost -> 0

A Simple GA

Generate a population with N random individuals.

WHILE not happy enough with the solution quality DO

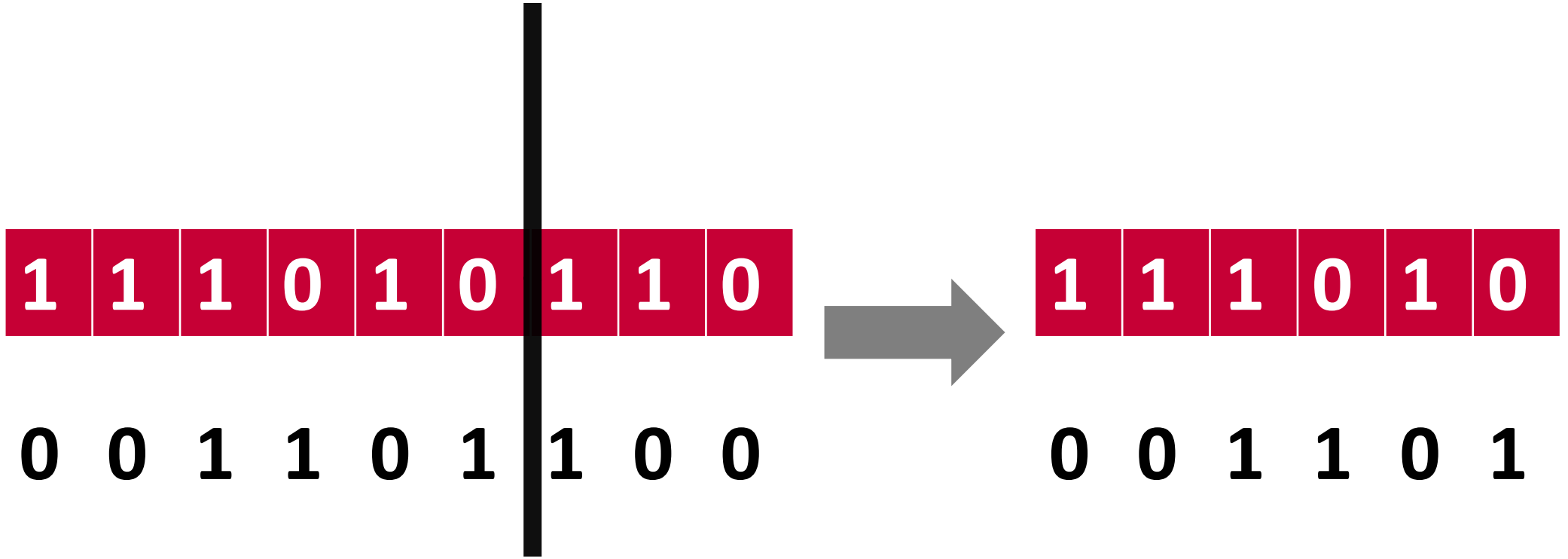
- Compute the fitness of every individual in the population
- Select better individuals
- Do crossover between pairs of selected individuals
- Mutate each gene with probability Pm

END WHILE

Crossover

- Recombine 2 individuals (father and mother) to obtain two new individuals (the children)
- Dedicated operators may be needed to ensure chromosome remains a valid solution

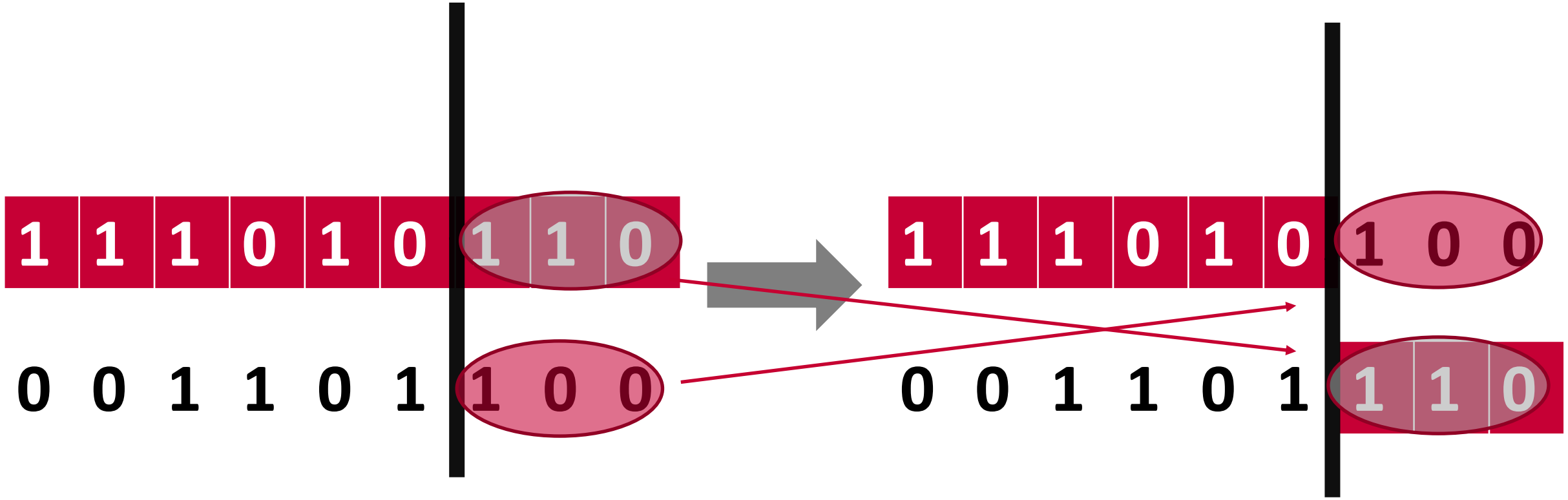
Example of crossover



Before recombination

After recombination

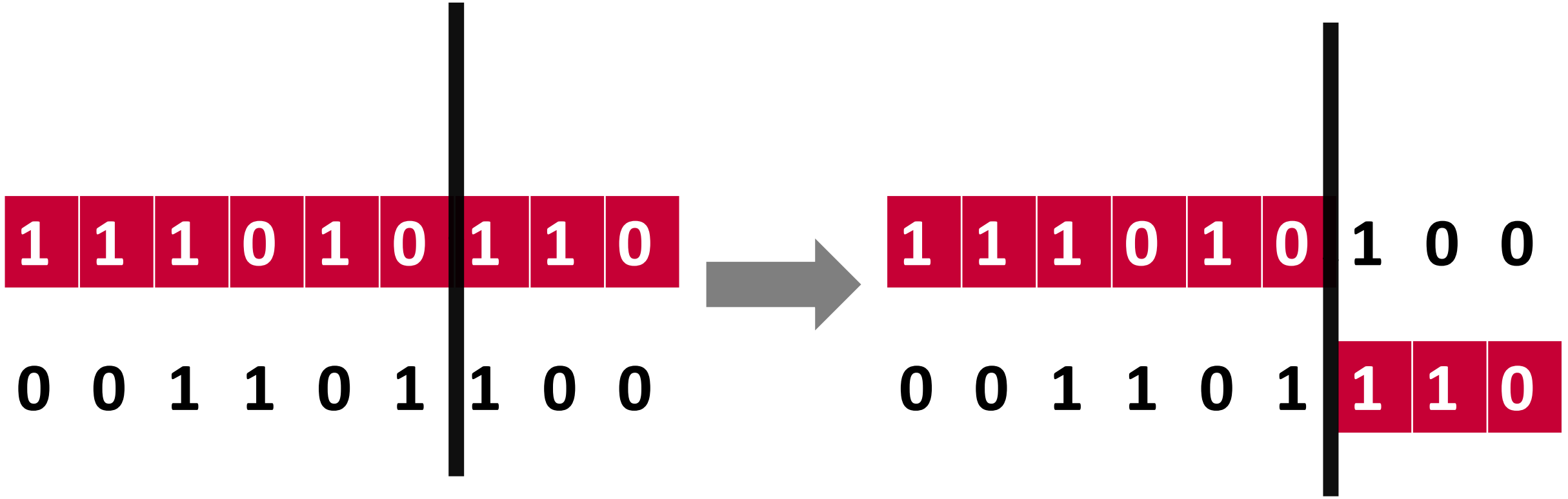
Example of crossover



Before recombination

After recombination

Example of crossover



Before recombination

After recombination

Mutation

- With probability Pm, flip a gene from 0 to 1, or from 1 to 0
- In traditional GAs, this operator is typically used with a low probability
- Low probability but important!

Gene 7 was mutated

Encodings of combinatorial problems

- Repetition is allowed (e.g., SAT Problem)
  - When the codification has ***n*** different types. We have ***n*** choices each time
    - First variable: n possible values
    - Second variable: n possible values
    - Third variable: n possible values
    - And so on
- Without Repetition (e.g., TSP Problem)
  - In this case, we have to reduce the number of available choices each time
    - First variable: n possible values
    - Second variable: n-1 possible values
    - Third variable: n-2 possible values
    - And so on

Popular Crossover operators

- One-point crossover
- Two-point crossover
- K-point crossover
- Uniform crossover

→ **Repetition is allowed**

- Uniform order based crossover
- PMX crossover
- Cycle crossover

→ **Without Repetition**

Mutation for permutation problems

- Performed per **chromosome** rather than per gene

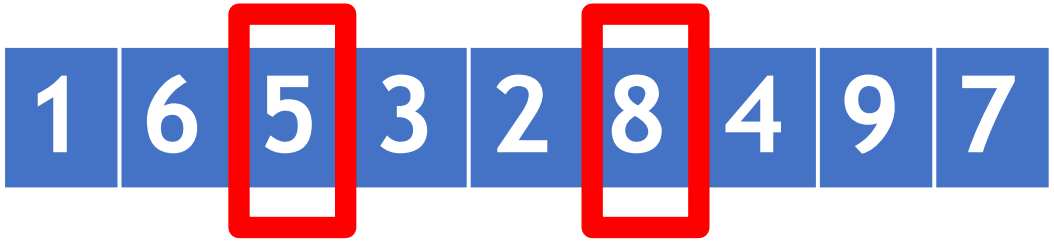
- Mutation rate = 0.1 => 1 in 10 children will be mutated on average

```
def mutation(self, ind):
    """
    Mutate an individual by swaping two cities with certain probability
    """
    if random.random() > self.mutationRate:
        return
    indexA = random.randint(0, self.genSize-1)
    indexB = random.randint(0, self.genSize-1)

    tmp = ind.genes[indexA]
    ind.genes[indexA] = ind.genes[indexB]
    ind.genes[indexB] = tmp

    ind.computeFitness()
    self.updateBest(ind)
```

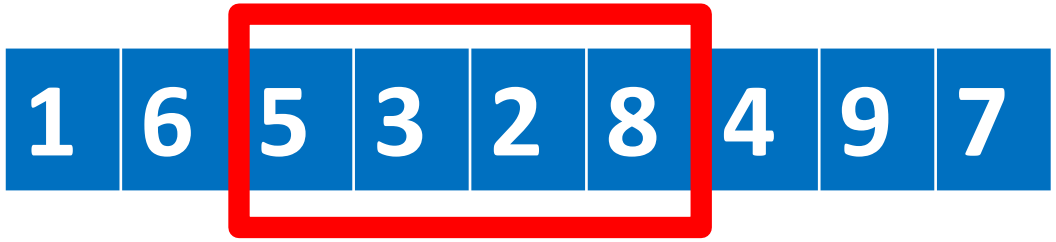
Mutation operators for permutation



Reciprocal Exchange

1 6 8 3 2 5 4 9 7

Mutation operators for permutation



Inversion Operator

1 6 8 2 3 5 4 9 7

Selection

- Survival of the fittest

- Bias selection towards fitter parents
- Populate mating pool based on fitness of chromosomes
  - Fitter chromosomes may occur multiple times in mating pool, increasing the chance of selection
  - Weaker parents may die and not be included in the mating pool.

- Proportionate and Ordinal

Proportionate Selection

- Sum fitness values across chromosomes to get total fitness

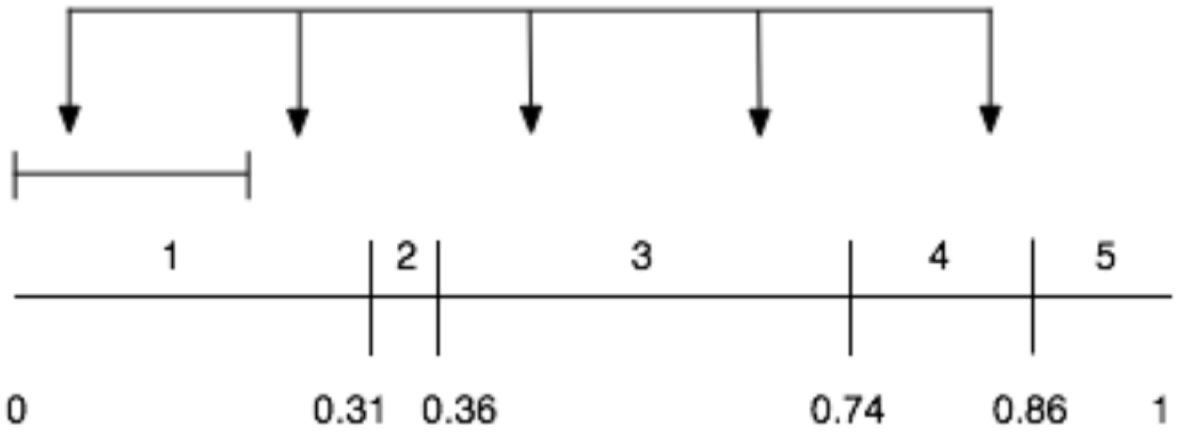
- Probability of selection for a chromosome is its fitness / total fitness

$$p_{select}(X) = \frac{f(X)}{\sum_{Y \in P(t)} f(Y)}$$

- For minimization?
  - Transform fitness first, e.g. 1/fitness or Max-fitness

Stochastic Universal Sampling (SUS)

- Populate mating pool using this rather than
- Make N equally spaced marks
- Generate a uniform random number in [0, 1/N]. That number is the position of the 1st mark.
- (Equivalent to spinning the equally spaced marks on top of the roulette. Just need to spin it once)



Ordinal-base methods

- Probability of selecting an individual depends on its relative order (or ranking) compared with other members

- Two commonly used methods:

  - Tournament selection

  - Truncation selection

Proportionate vs. ordinal-based methods

- Fitness proportionate-based methods have 2 major drawbacks:
  - Super individual can take over the population too quickly
  - Search tends to stall when all the individuals have about the same fitness
- Ordinal-based methods are often preferred
  - Enable a sustained pressure towards better solutions
  - Can control selection pressure easily

Replacement

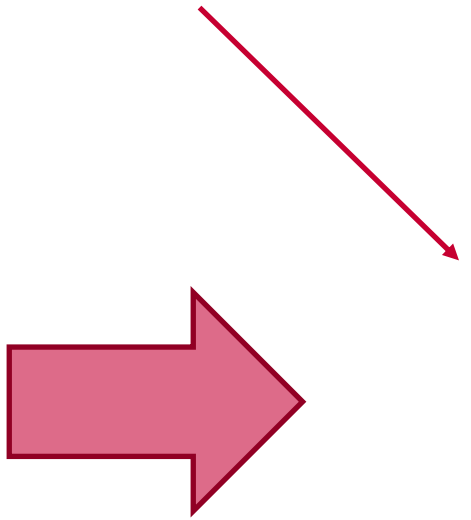
- Full Replacement
- Elitist: Don’t throw away the best solutions
  - If the fitness of the best from the previous generation is better than the worst in the new generation, replace the worst

Elitist

Best Candidates
Or Elite Candidates

	Generation at time t
1	
2	
N-1	
N	

Current Generation



	Generation at time t+1
1	
2	
N-1	
N	

New Generation

Swarm Intelligence: ACO

- Basic concept
  - Mimic indirect cooperation behavior of ants in finding shortest path to food source from nest.
  - Ants leave pheromone trail on path back to nest, subsequent ants will be attracted to path with most pheromones, pheromones evaporate over time.
- ACO
  - Create a set of artificial ants (same idea as population in GA)
  - Each ant will find a path through the search space such that it generates a candidate solution
  - After all ants have found a solution, they return to the start, and deposit “pheromones” on the path they followed; pheromone function of the solution quality
  - Next iteration, ants choose next node with probability based on pheromone on edge between nodes.
  - Variants

Greedy Best First Search

- Heuristic to generate a solution.
- Start from random point, choose neighbour that has best *heuristic* value, i.e. is estimated to be best of the options from the perspective of the objective value.
- Can get stuck in infinite loops if previous states not stored in a *tabu*-type list

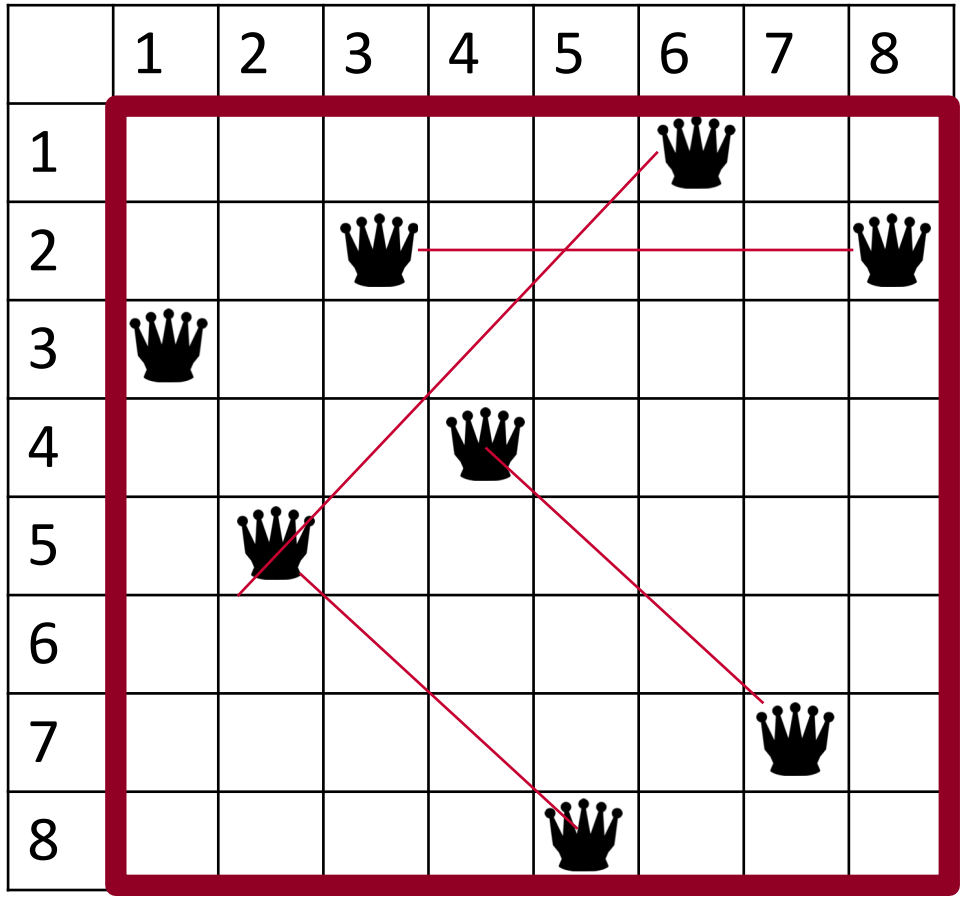
Local Search

- Iteratively improve a single solution
- Each iteration we move from the current state to a *neighbor* state
- Possible moves restricted to neighborhood

Recap: Neighborhoods

- Key part of designing of you local search algorithm
- Often involve changing value of 1 variable in solution
- Examples:
  - N-queens: Move a queen to a different row
  - SAT: *Flip* a variable, i.e. change its value from true to false or false to true
  - TSP: Swap two cities in route
- Neighborhood depends on representation!

Representation



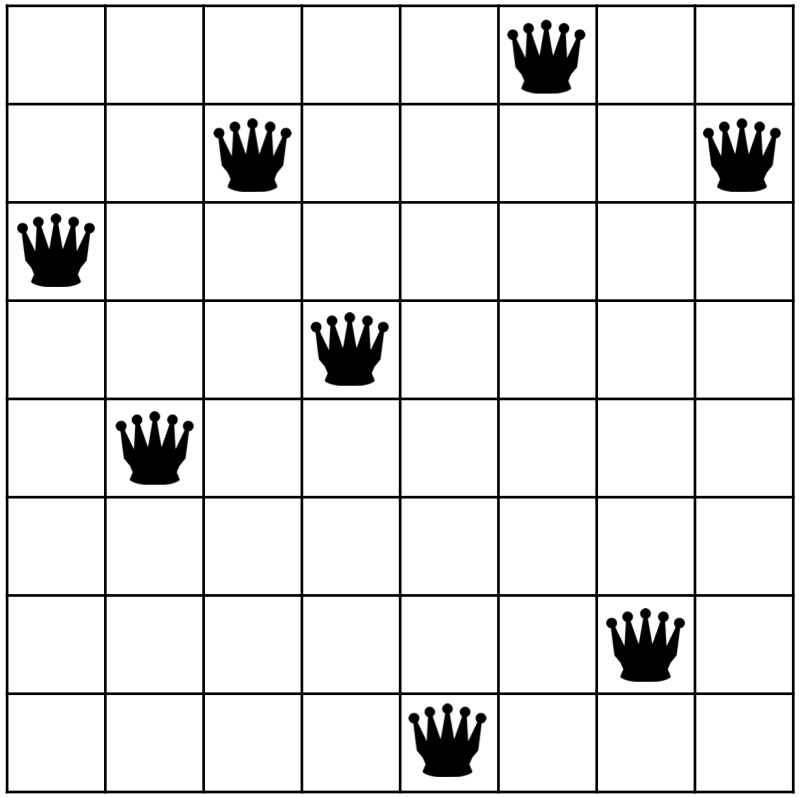
3, 5, 2, 4, 8, 6, 7, 8

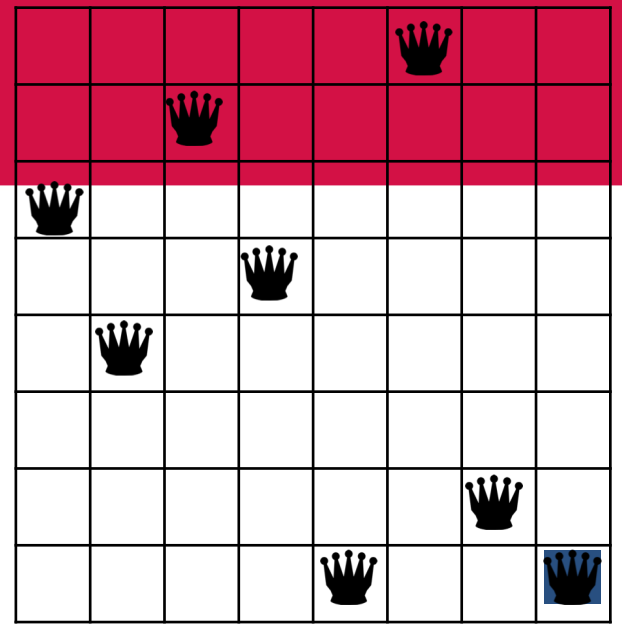
Pair of Attacking queens:

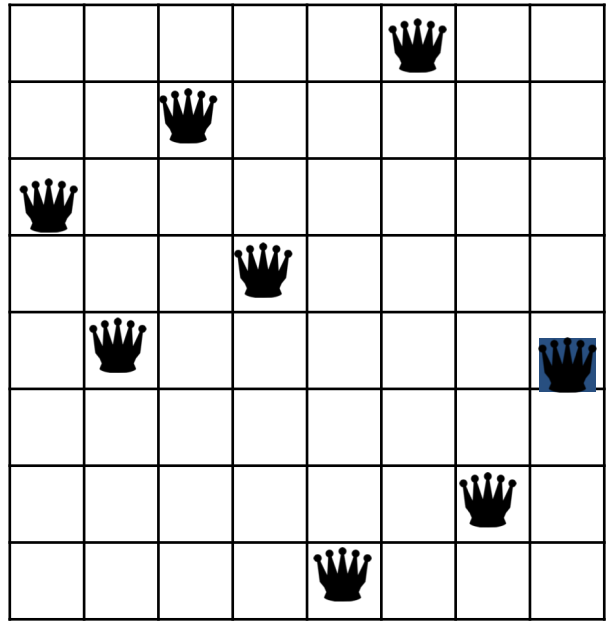
?

4

Neighbors: N-queens

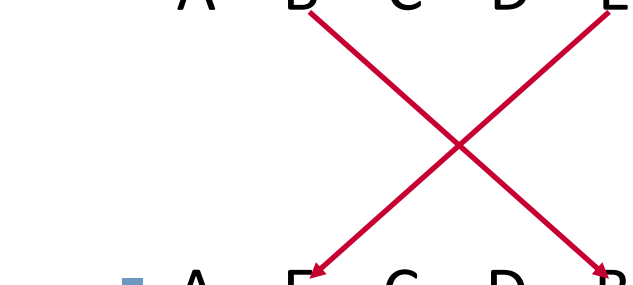




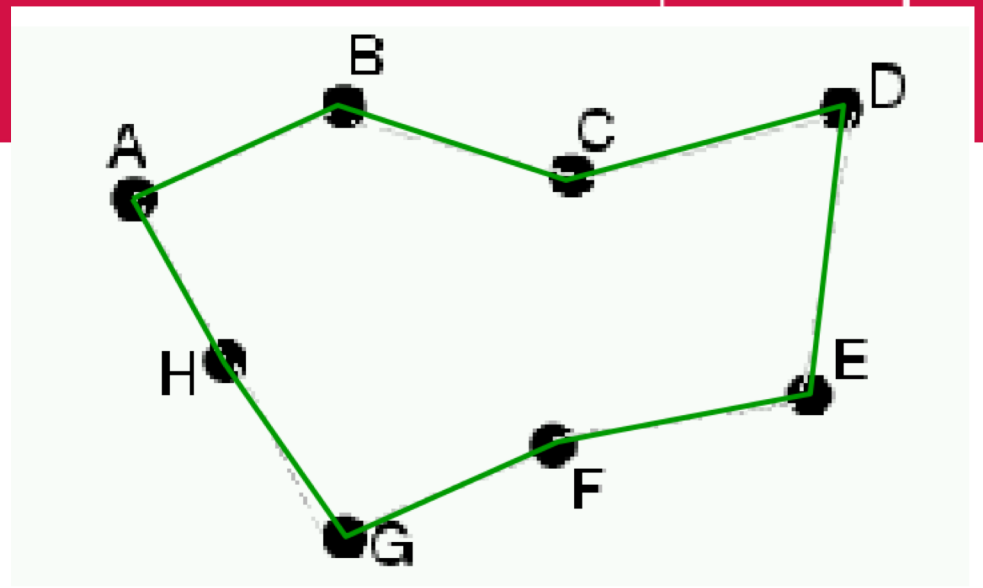


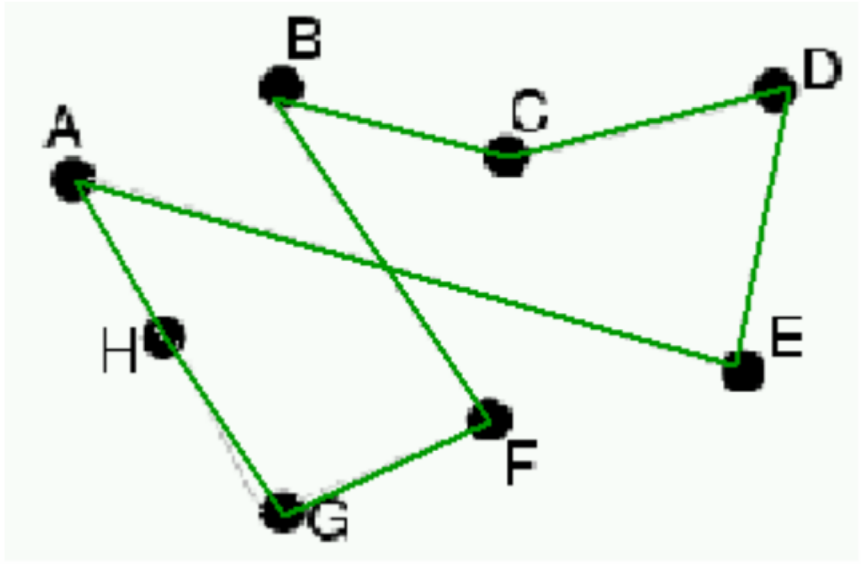
Neighbors: TSP

- State: A – B – C – D – E – F – G – H – A
- One possibility: 2-change
- A – B – C – D – E – F – G – H – A



- A – E – C – D – B – F – G – H – A





k-SAT

- **Subclass of the SAT problem**
  - Exactly k variables in each clause

    $(P \vee \neg Q) \wedge (Q \vee R) \wedge (\neg R \vee \neg P)$ is in 2-SAT

- **3-SAT is NP-complete**
  - SAT can be reduced to 3-SAT

- **2-SAT is in P**
  - Exists linear time algorithm

Neighbors: SAT

- State: (A=T, B = F, C=T, D=T, E=T)
- Neighbor: Flip the assignment of one variable

- (**A=F**, B = F, C=T, D=T, E=T)
- (A=T, **B = T**, C=T, D=T, E=T)
- (A=T, B = F, **C=F**, D=T, E=T)
- (A=T, B = F, C=T, **D=F**, E=T)
- (A=T, B = F, C=T, D=T, **E=F**)

$$A \vee \neg B \vee C$$
$$\neg A \vee C \vee D$$
$$B \vee D \vee \neg E$$
$$\neg C \vee \neg D \vee \neg E$$
$$\neg A \vee \neg C \vee E$$

Hill-climbing (Greedy LS) Max Version

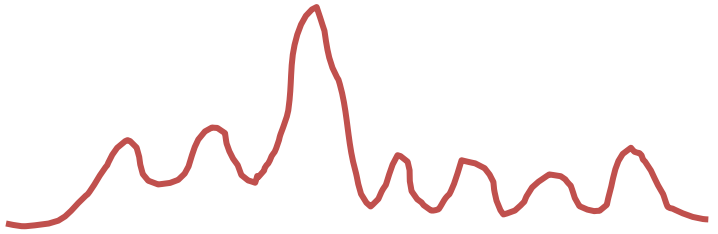
function HILL-CLIMBING(*problem*) **return** a state that is a local maximum
input: *problem*, a problem
local variables: *current*, **a node.**
neighbor, **a node.**

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*

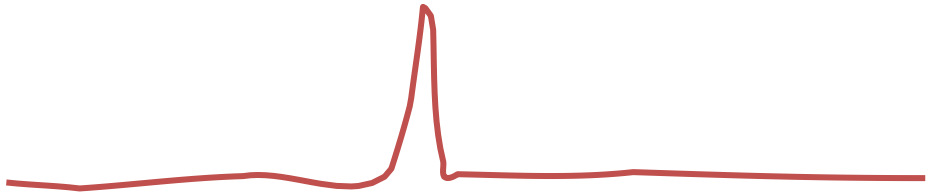
Min version will reverse inequalities and look for lowest valued successor

Hill Climbing Drawbacks

Local maxima



Plateaus



Recap: Hill-climbing search: 8-queens problem

- Randomly generated 8-queens starting states…
  - 14% of the time it solves the problem
  - 86% of the time it get stuck at a local minimum

- However…
  - Takes only 4 steps on average when it succeeds
  - And 3 on average when it gets stuck

    (for a state space with 8^8 =~17 million states)

Hill climbing & SAT → GSAT

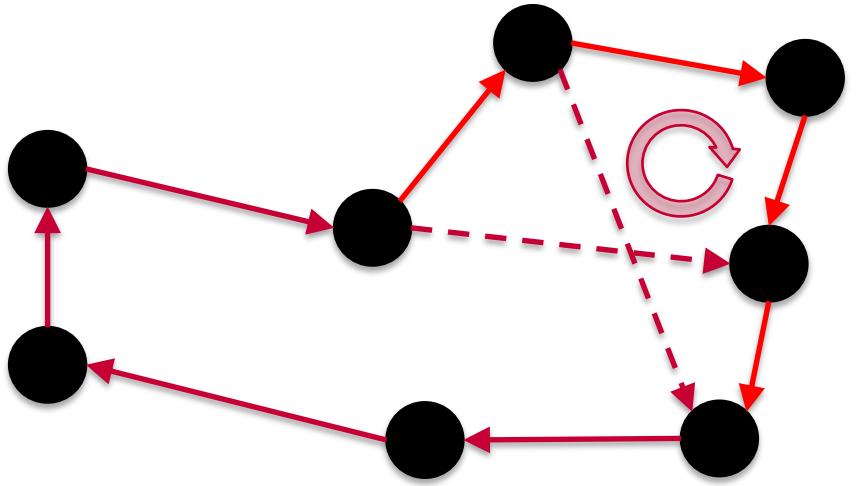
```
procedure GSAT (F, maxSteps)
    input CNF formula F, positive integers maxTries and maxSteps
    output model of F or "no solution found"
     a := randomly chosen assignment of the variables in formula F;
     for step := 1 to maxSteps do
         if a satisfies F then return a;
         v := randomly selected variable the flip of which minimises
              the number of unsatisfied clauses;
         a := a with v flipped;
     end for;
    return "no solution found";
end GSAT
```

Local Search for SAT

- Choose a **random initial solution** *I*
  - Random values for the variables (0 or 1 to all variables)
- If *I* is not a valid solution, repeatedly choose a variable p and **change its value** in *I* (flip the variable)
  - Flip determines the successor state
- Flipped variables are chosen using heuristics or randomly or both
  - The evaluation function is usually the number of satisfied clauses in a state
- Most algorithms use random restarts if no solution has been found after a fixed number of search steps

Local search for TSP: 2-OPT, 3-OPT, 4-OPT, k-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select two edges and replace them by two other edges
  - Select three edges and replace them by three other edges



Recap: Importance of modeling choices!

- Warehouse location provlem: where to place warehouses? (input: f_w , $t_{w,c}$)

- Decision Variables:
  - $o_w \in \{0, 1\}$: whether warehouse w is open
  - $\alpha[c] \in \{1, \ldots, W\}$: the warehouse assigned to customer c

- Objective $\min \sum_{w \in W} f_w o_w + \sum_{c \in C} t_{a[c],c}$

- Key observations:
  - Once the warehouse locations have been chosen, the problem is easy
  - It's enough to assign customers to warehouses minimizing the transportation costs
  - **Note the difference between testing different values for the o_w variables and the $\alpha[c]$ variables**

Tabu Search

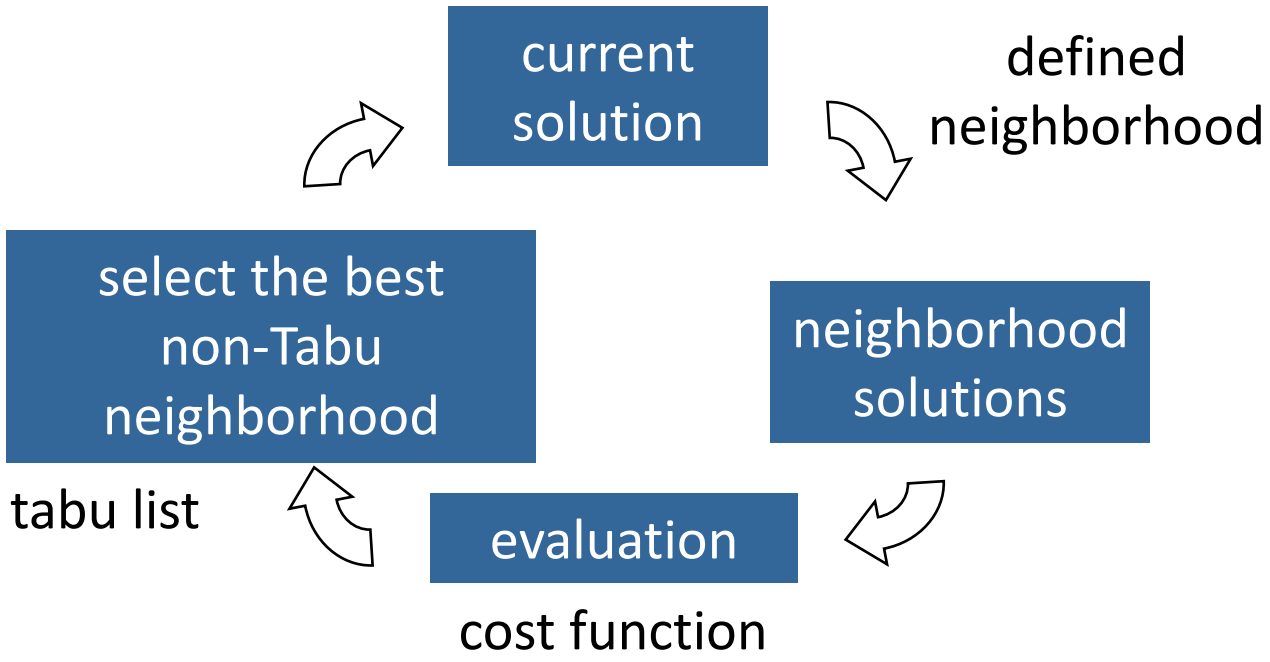
Tabu Search (TS):

determine initial candidate solution s
While *termination criterion* is not satisfied:
  determine set N' of non-tabu neighbours of s
  choose a best improving candidate solution s' in N'
  update tabu attributes based on s'
  $s := s'$

Tabu search prevents returning quickly to same state.

- To implement: Keep fixed length queue (tabu list).
- Add most recent step to queue; drop oldest step.
- Never make step that's on current tabu list.

Tabu Search


current solution
defined neighborhood
neighborhood solutions
evaluation
cost function
select the best non-Tabu neighborhood
tabu list

Recap: Tabu Search Efficiency

- Instead of storing a queue that needs to be searched for each candidate variable, store the lowest iteration number that this variable may be flipped again after

- Reduces from linear to constant time

Iterative Local Search

Iterated Local Search (ILS):

determine initial candidate solution s

perform *subsidiary local search* on s

While termination criterion is not satisfied:

- $r := s$
- perform *perturbation* on s
- perform *subsidiary local search* on s
- based on *acceptance criterion*,
  - keep s or revert to $s := r$

Dynamic Local Search

Dynamic Local Search (DLS):

determine *initial candidate solution* s

initialise penalties

While *termination criterion* is not satisfied:

- compute *modified evaluation function* g' from g based on *penalties*
- perform *subsidiary local search* on s using *evaluation function* g'
- *update penalties* based on s

Hill-climbing: stochastic variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.

- To avoid getting stuck in local minima
  - Random-walk hill-climbing
  - Random-restart hill-climbing
  - Hill-climbing with both

Simulated annealing

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

input: *problem*, a problem

schedule, a mapping from time to temperature

local variables: *current*, a node.

next, a node.

T, a "temperature" controlling the 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 $e^{\Delta E / T}$

Stopping Conditions

- **No feasible solution** in the neighborhood of solution i
- The number of iterations since the last improvement of i* is larger than a specified number
- Evidence can be given than an **optimum solution** has been obtained

Search Process

- **Intensification**:
  - searches solutions **similar** to the current solution
  - wants to **intensively** explore known promising areas of the search space
  - creates solutions using the more *attractive* **components** of the best solutions in memory
  - Another technique consists in changing the neighborhood's structure by allowing different moves
- **Diversification**:
  - Examines **unvisited** regions of the search space
  - Generates different solutions
    - Using rarely components present in the current solution
    - Biasing the evaluation of a move by modifying the objective function adding a term related to component frequencies
- Intensification and diversification phases **alternate** during the search