Methods of Artificial Intelligence

2. Local Search Algorithms

Nohayr Muhammad

Winter Term 2022/2023

November 11th, 2022

Last time..

- Solving problems by searching
- Search problems
- (Classical) Search algorithms
- Searching in complex environments
- Local search main ideas

Today..

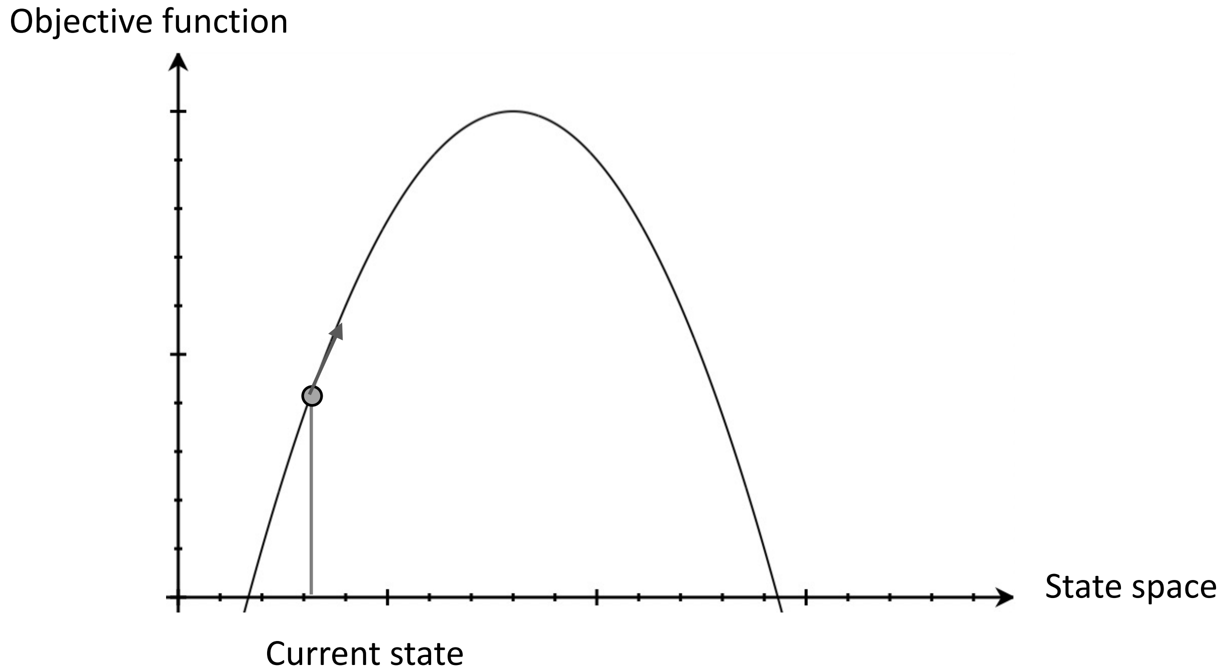
- Hill-Climbing
- Simulated Annealing
- Local Beam Search
- Genetic Algorithms

State-space Landscape and Hill-Climbing

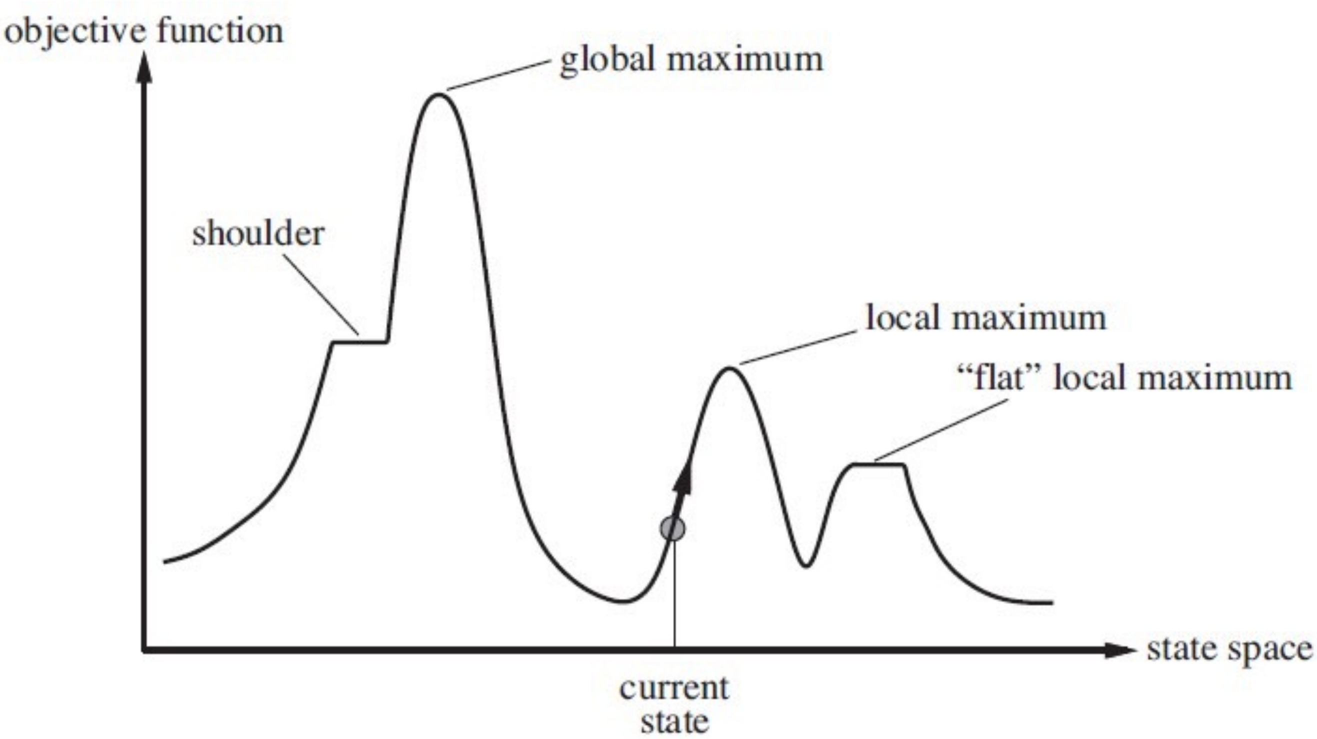
What is the simplest form of local search?

Gradient Ascent Revisited

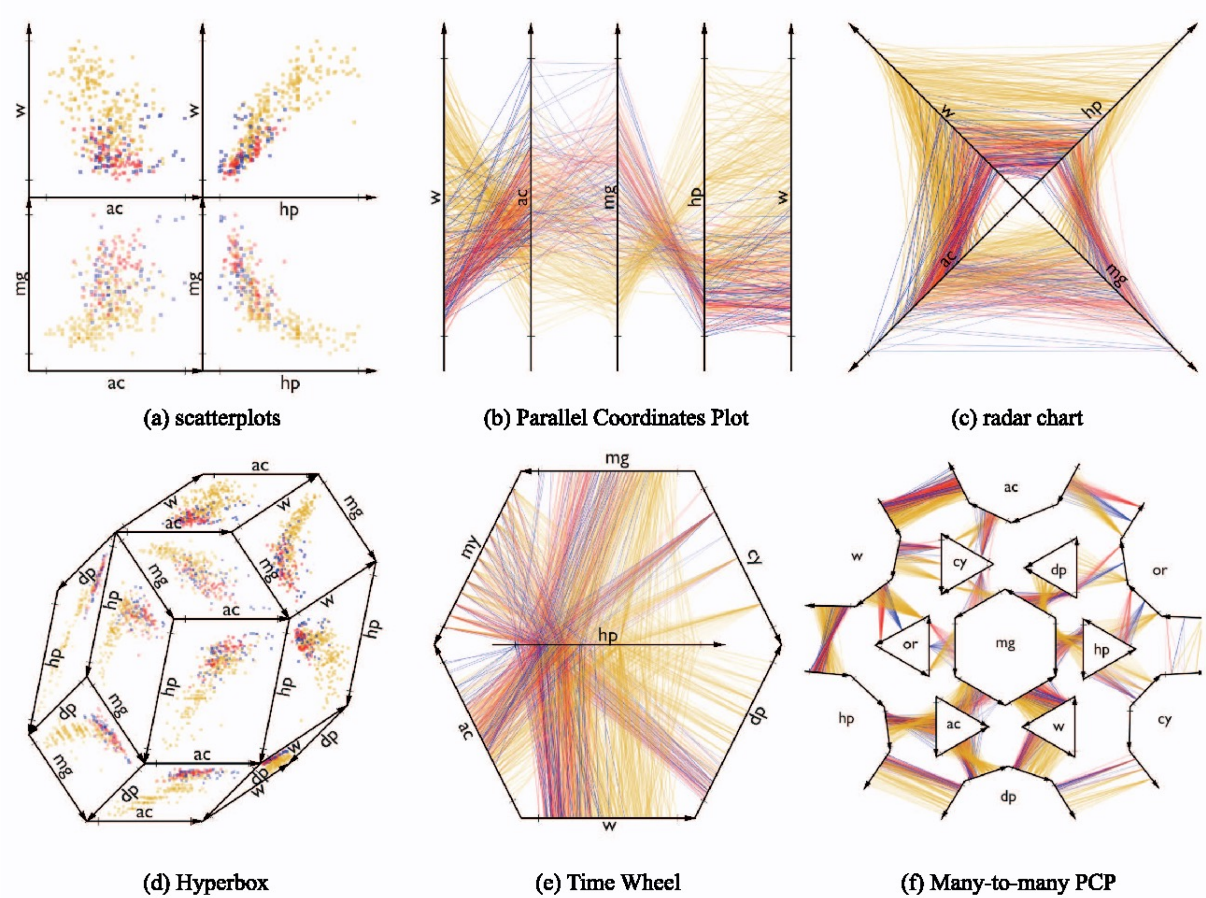
- Suppose, we want to maximize real function f(x) (objective function)
- we can do so by starting from random point and following ascent direction (gradient ascent algorithm)



It’s a little more complicated



Actually, it’s much more complicated

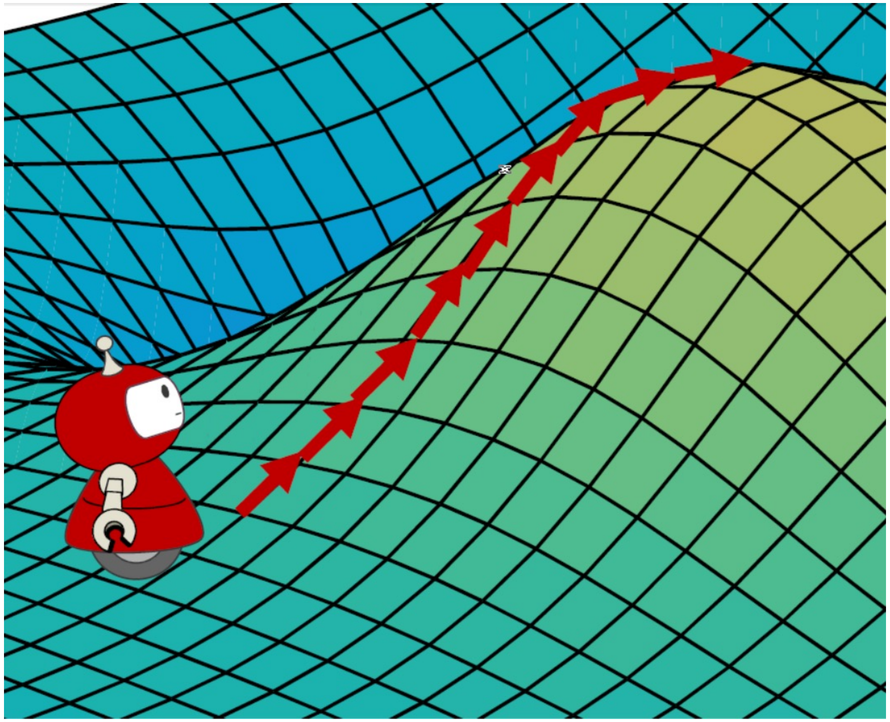


(a) scatterplots (b) Parallel Coordinates Plot (c) radar chart (d) Hyperbox (e) Time Wheel (f) Many-to-many PCP

[Flexible Linked Axes for Multivariate Data Visualization]

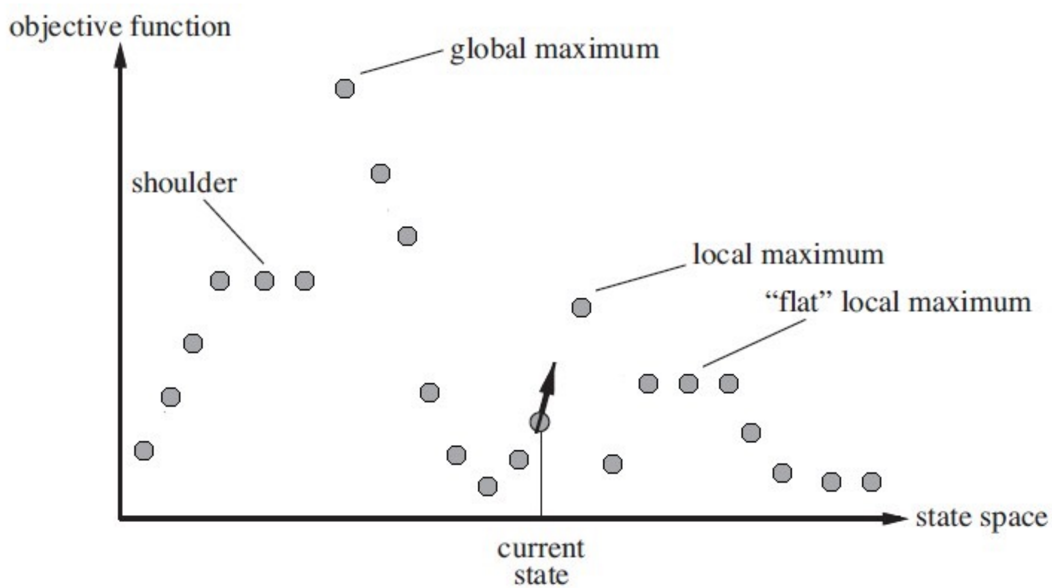
Hill-Climbing

- Hill-Climbing can be regarded as a discrete state-space version of gradient ascent
- In each step, we select neighbor with highest value



https://www.mathworks.com/matlabcentral/fileexchange/74015-hill-climbing-algorithm-a-simple-implementation

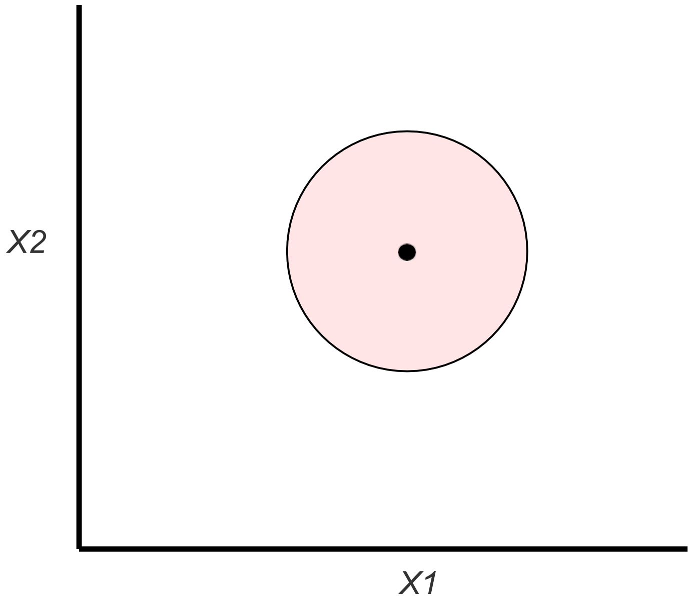
Discrete State Spaces



- We will often consider finite state spaces here
- Problem: state space is usually exponentially large

Continuous vs. Discrete Spaces

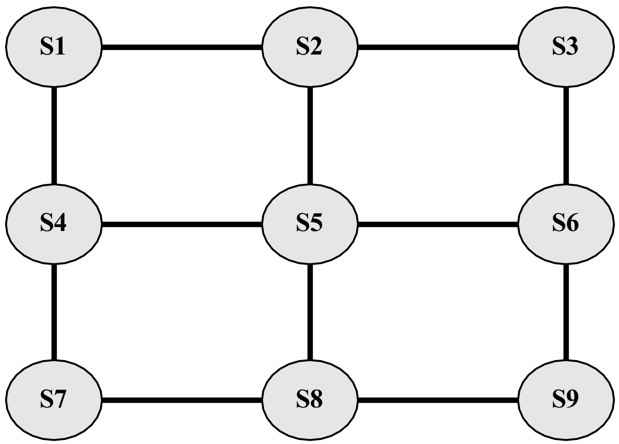
Continuous Space



ε-neighborhood

- Infinite number of neighbors
- Gradient gives direction of steepest ascent

Discrete Space



Discrete neighborhood

- Often finite number of neighbors
- Best neighbor can be difficult to find (enumeration)

Hill-Climbing Algorithm

current ← select random initial state
do
  neighbor ← neighbor of *current* with highest value
  if value(*neighbor*) ≤ value(*current*)
    return *current*
  current←*neighbor*
until termination condition is met

Hill-Climbing: Termination Conditions

current ← select random initial state
do
  neighbor ← neighbor of *current* with highest value
  if value(*neighbor*) ≤ value(*current*)
    return *current*
  current←*neighbor*
until termination condition is met

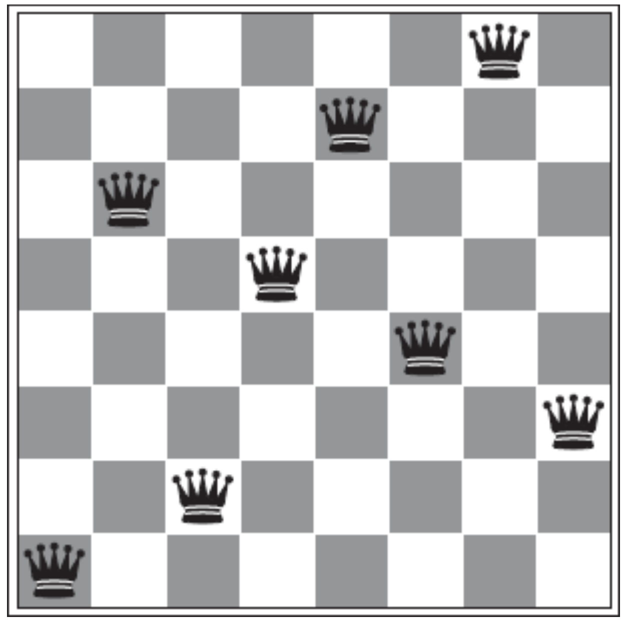
- termination condition can bound the maximum number of search steps or search time
- algorithm may end up in non-global local maximum or plateaux

Solving local search problems: Hill Climbing

1. Define state space
2. Define neighborhood
3. Define objective function

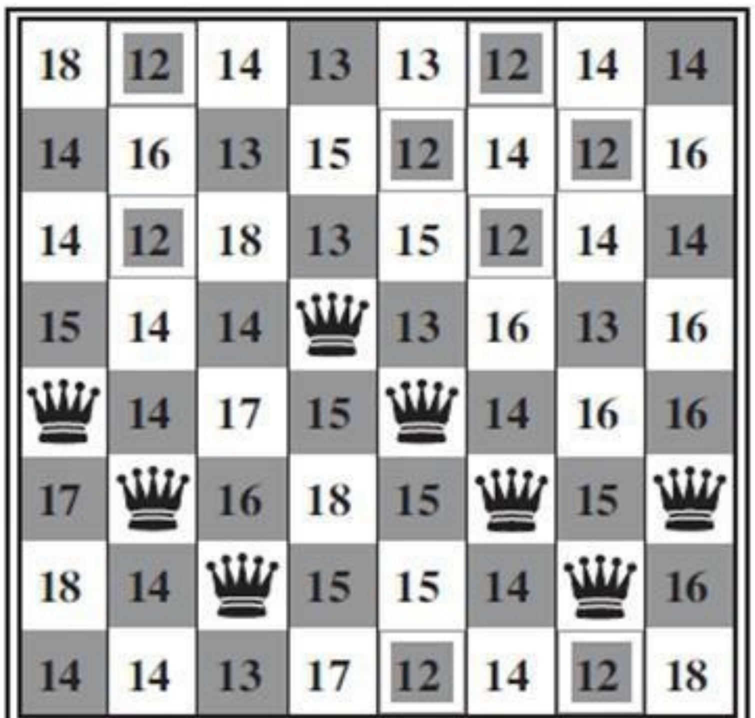
   (minimize f(x) by maximizing –f(x))
4. Apply hill climbing algorithm

Recall: 8-Queens Problem



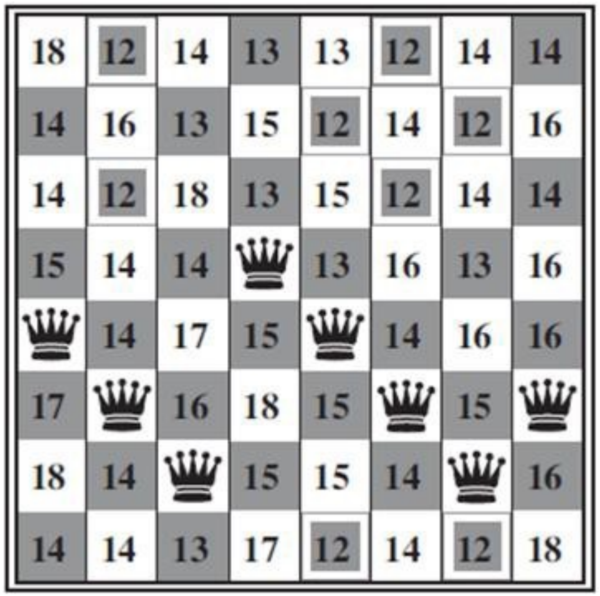
1. State-Space: board configurations with one queen per column (tuple (8,3,7,4,2,5,1,6) corresponds to board configuration above).
2. Neighborhood: configurations that can be obtained by moving a single queen to another field in the same column.
(hence, every state has 8 * 7 = 56 neighbours)

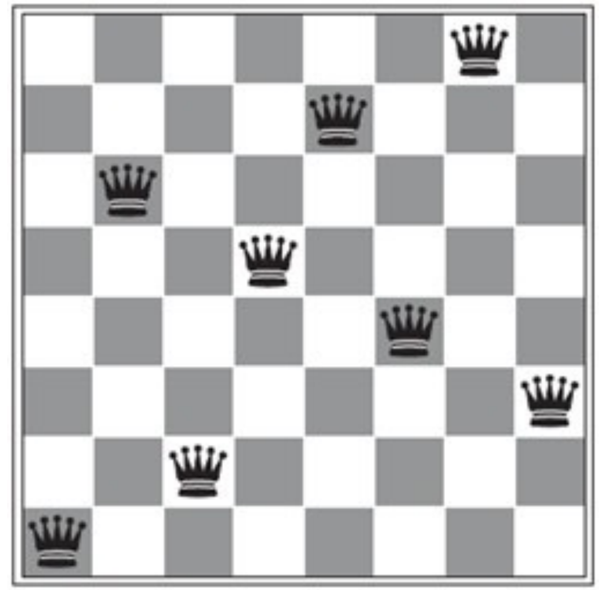
3. **Objective Function:** number of pairs of queens that can attack each other directly or indirectly *(maximize negative objective function)*



- Queen in column 1 can directly attack queen in column 2
- Queen in column 1 can indirectly attack queen in column 3
- State has value 17
- Numbers show values of neighbors

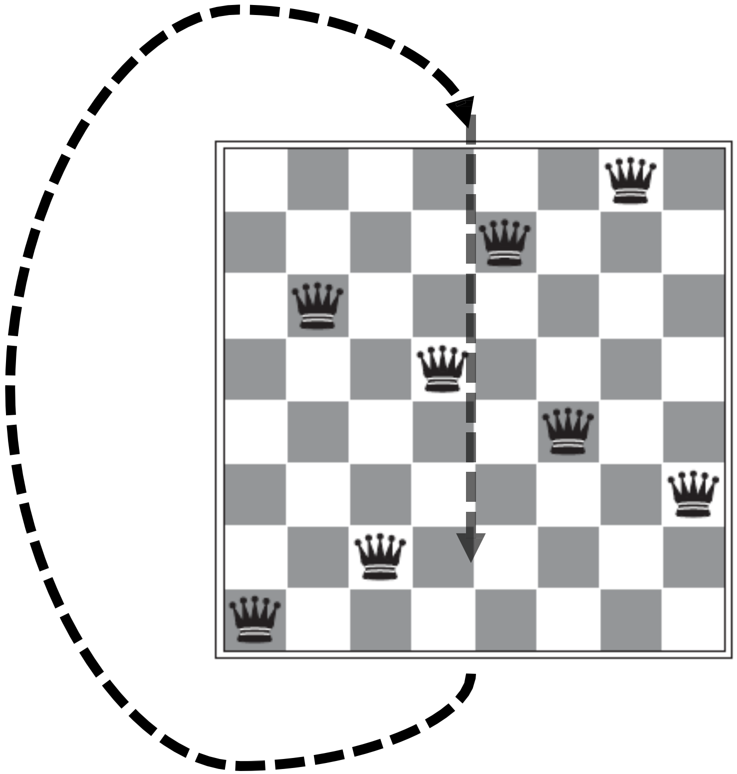
Local Optima





- Hill Climbing goes from left state with value -17 to right state with value -1 (queen 4 attacks queen 7) in only 5 steps
- However, right state is only locally optimal

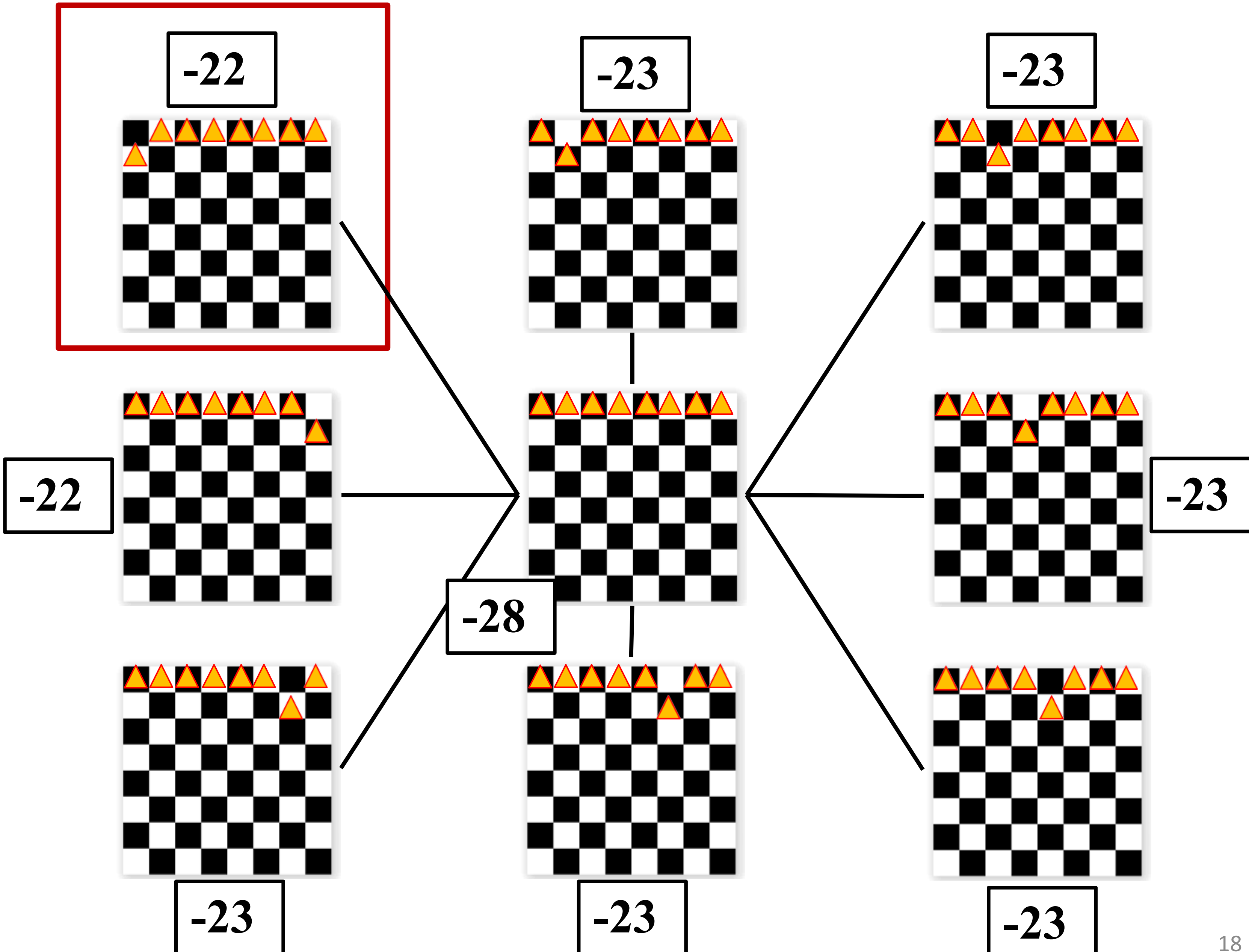
A smaller neighborhood



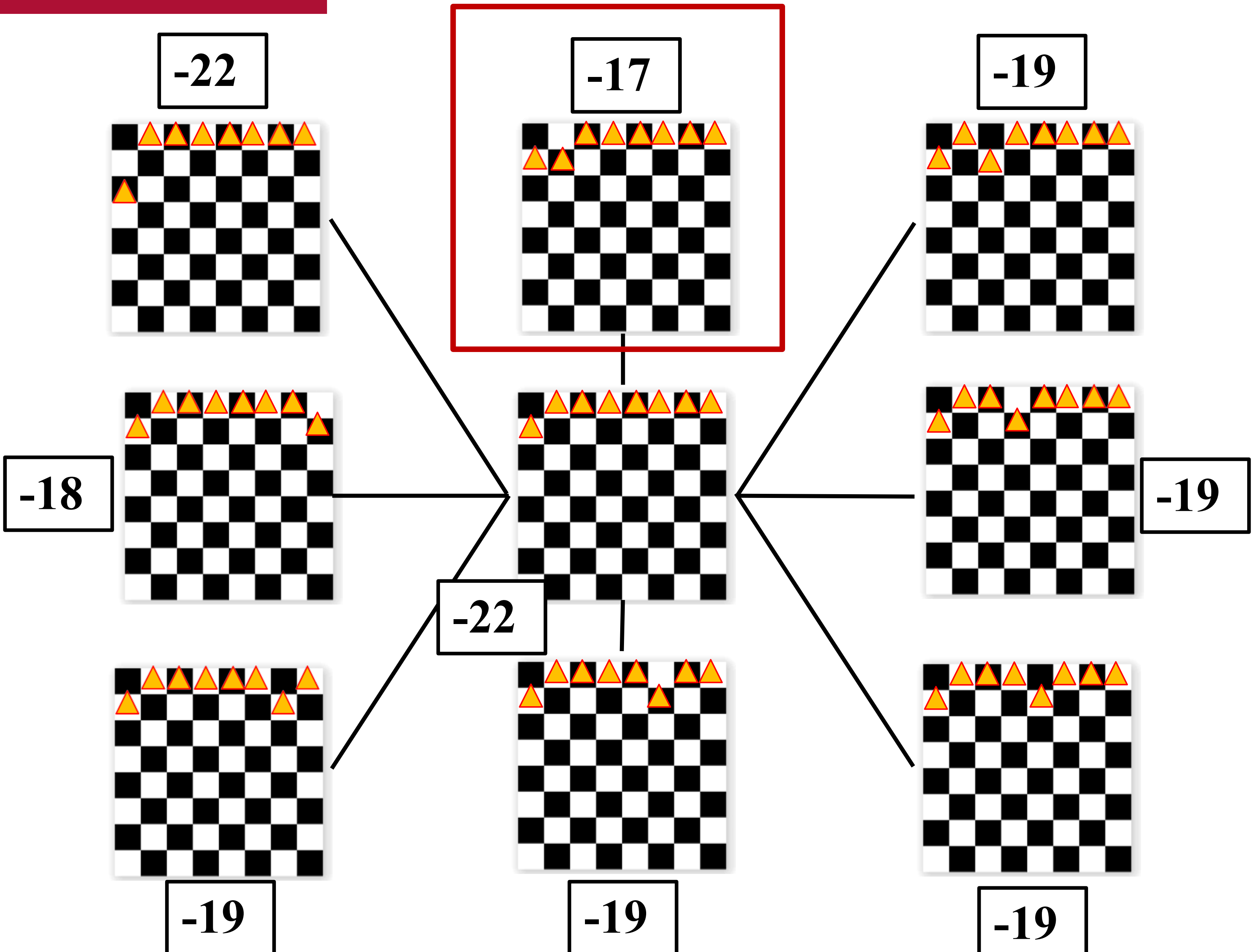
- Neighborhood: configurations that can be obtained by moving a single queen down by one field (toroidal board)

(hence, every state has 8 neighbors)

Example



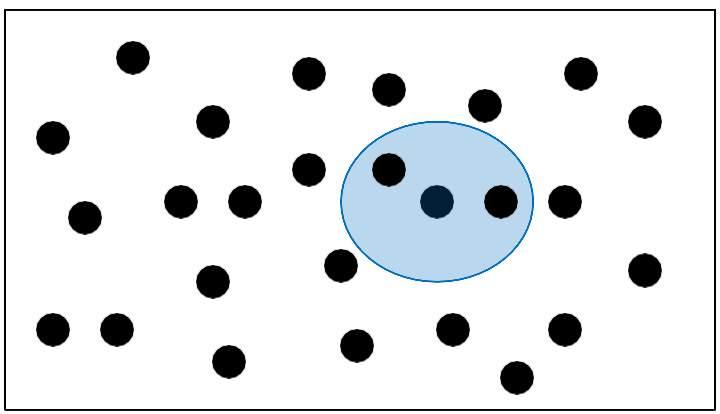
Example



Tradeoff

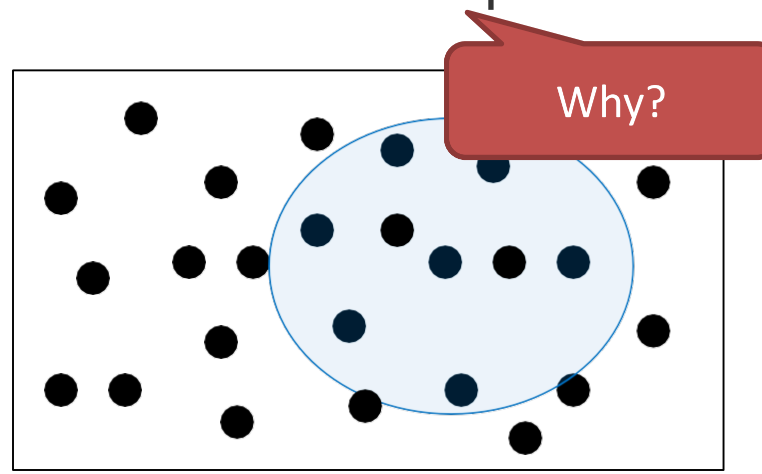
Small Neighborhood

- Faster exploration of neighbord
- Greater risk of getting stuck in local optimum

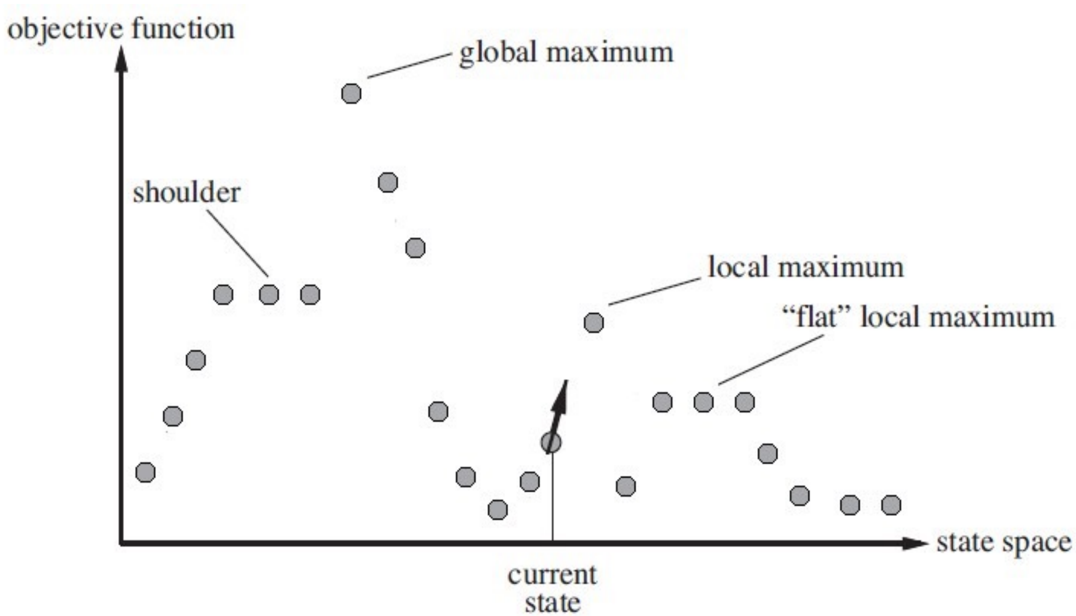


Large Neighborhood

- Slower exploration of neighborhood
- Smaller risk of getting stuck in local optimum



Variants of Hill-Climbing



- Hill-Climbing in its basic form can perform poorly because there is a high risk of ending up in non-global local maxima
- There exist several variants that can alleviate the problem

Stochastic Hill-Climbing

```
current ← select random initial state
do
   neighbor ← random neighbor of current with higher value
   if neighbor = null
      return current
   current←neighbor
until termination condition is met
```

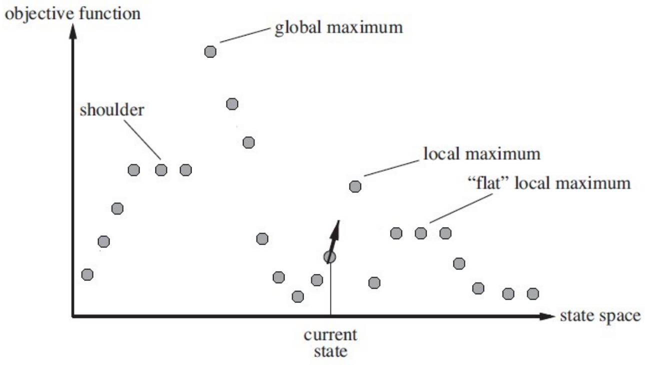
- Instead of selecting neighbor with maximum value, we select random neighbor that improves value
- Probability of selection can increase with value
- Compromise between random search and Hill-Climbing

Other Variants

- First-choice Hill-climbing: in neighbor selection step, pick first neighbor that improves objective function (also useful if neighborhood is too large to enumerate all neighbors)
- Random-restart (or parallel) Hill-Climbing: perform *n* independent

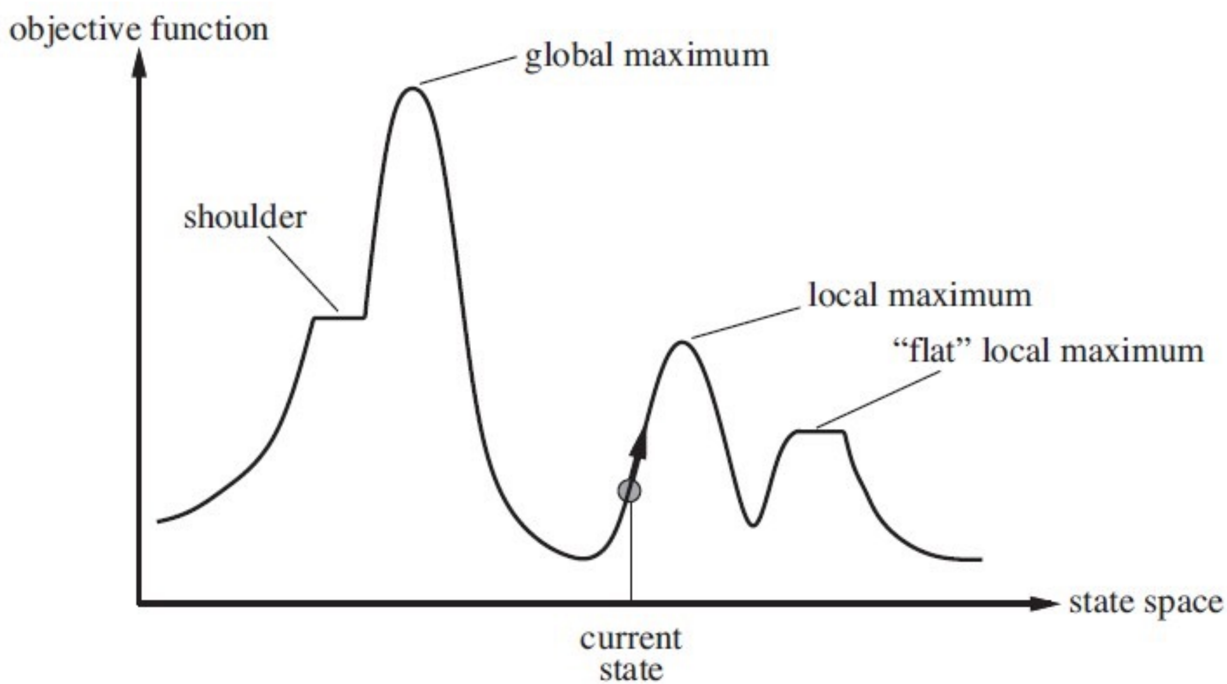
  Hill-climbing searchs starting from randomly generated initial states

  (as *n* goes to infinity, probability of finding global optimum goes to 1)



Simulated Annealing

Downhill Moves



- Hill-climbing never makes downhill moves
- However, downhill moves can be necessary to find global optimum

Simulated Annealing Intuition

- Initially: large probability for downhill moves (exploration)
- As search progresses: probability decreases (intensification)
- This process is modeled by means of a temperature variable that decreases (thus simulated annealing)

Simulated Annealing

```
current ← select random initial state
for t = 1 to ∞
   T ← schedule(t)
   if T = 0
      return current
   next ← randomly selected neighbor of current
   ΔE ←value(next) – value(current)
   if ΔE > 0
      current ←next
   else
      current ←next only with probability exp(ΔE/T)
```

Cooling Schedule

- schedule(t) controls temperature decrease
- Some simple scheduling schemes:
  - Stepwise Linear: start with arbitrary T and decrease T by a constant c in each step
  - Delayed Stepwise Linear: start with arbitrary T and decrease T by a constant c every k-th step
- A slow cooling schedule increases probability of finding a high-quality solution but increases runtime

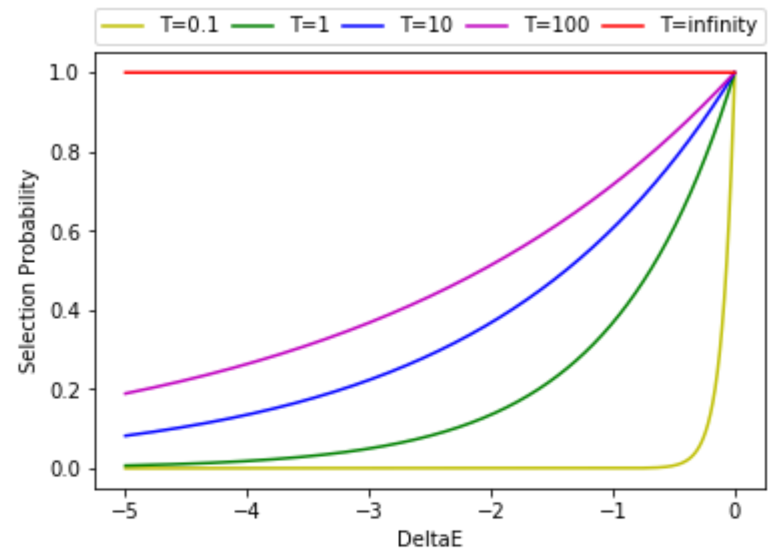
Neighbor Selection

- In each step, Simulated Annealing picks random neighbor
  - If neighbor improves objective, neighbor replaces current state
  - Otherwise, it replaces current state only with probability exp(ΔE/T), where ΔE = value(*next*) – value(*current*)

Neighbor Selection: ΔE

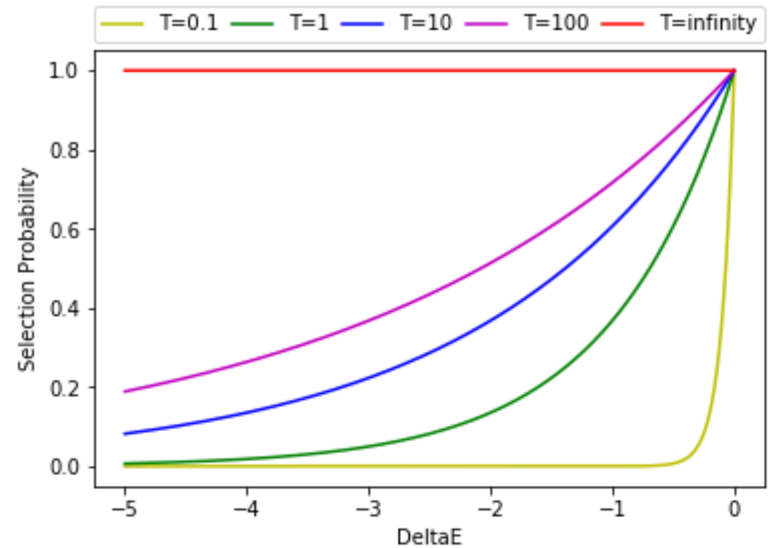
- ΔE is always non-positive in else-branch
- Hence, $0 \leq \exp(\Delta E/T) \leq \exp(0) = 1$

- For example, for **T=1**, we have
  - $\Delta E = 0 : \exp(0/1) = \exp(0) = 1$
  - $\Delta E = -1 : \exp(-1/1) = 1/e = 0.37$
  - $\Delta E = -2 : \exp(-2/1) = 1/e^2 = 0.14$



Neighbor Selection: Temperature

- As temperatue T decreases, probability decreases faster
- For example, assume **ΔE = -2**
  - T= ∞ : exp(-2/∞) = exp(0) = 1
  - T=100: exp(-2/100) = 0.98
  - T=10: exp(-2/10) = 0.81
  - T=1: exp(-2/1) = 0.13
  - T=0.1: exp(-2/0.1) = 0.002

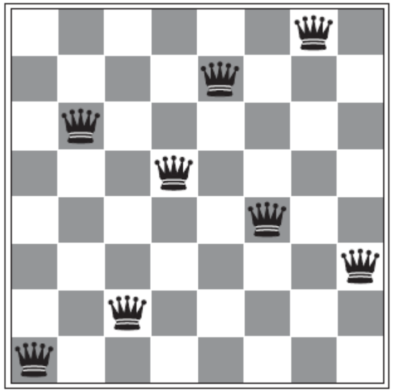


- Therefore, Simulated Annealing is similar to random exploration for high temperatures and similar to First-choice Hill-Climbing later

Implementing Neighbor Selection

- Naive implementation
  - Enumerate all neighbors and pick randomly
  - Runtime O(|Neighborhood|)
  - Reasonable when selection probability depends on value of state
  - Wasteful for uniform selection
- Uniform implementation
  - Create random neighbor
  - Runtime O(1)

Example: 8-Queens Problem



Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	Q[6]	Q[7]	Q[8]
8	3	7	4	2	5	1	6

- State-Space: board configurations with one queen per column
- Sampling
  1. Create random number q from {1,…,8} (select queen)
  2. Create random number from {1,…,8}\{Q[q]} (select new position)

Example Makespan Problem

J1	J2	J3	J4	J5	J6	J7	J8	J9	J10
M1	M2	M2	M1	M2	M3	M2	M3	M1	M1

- State Space: assignment of machines from {M1, M2, M3} to jobs
- Sampling

1. Create random number j from {1,…,10} (select job)
2. Create random number from {1,…,3}\{M[j]} (select new machine)

Example: Facility Location Problem

L1	L2	L3	C1	C2	C3	C4	C5	C6
0	0	1	1	2	3	2	1	3

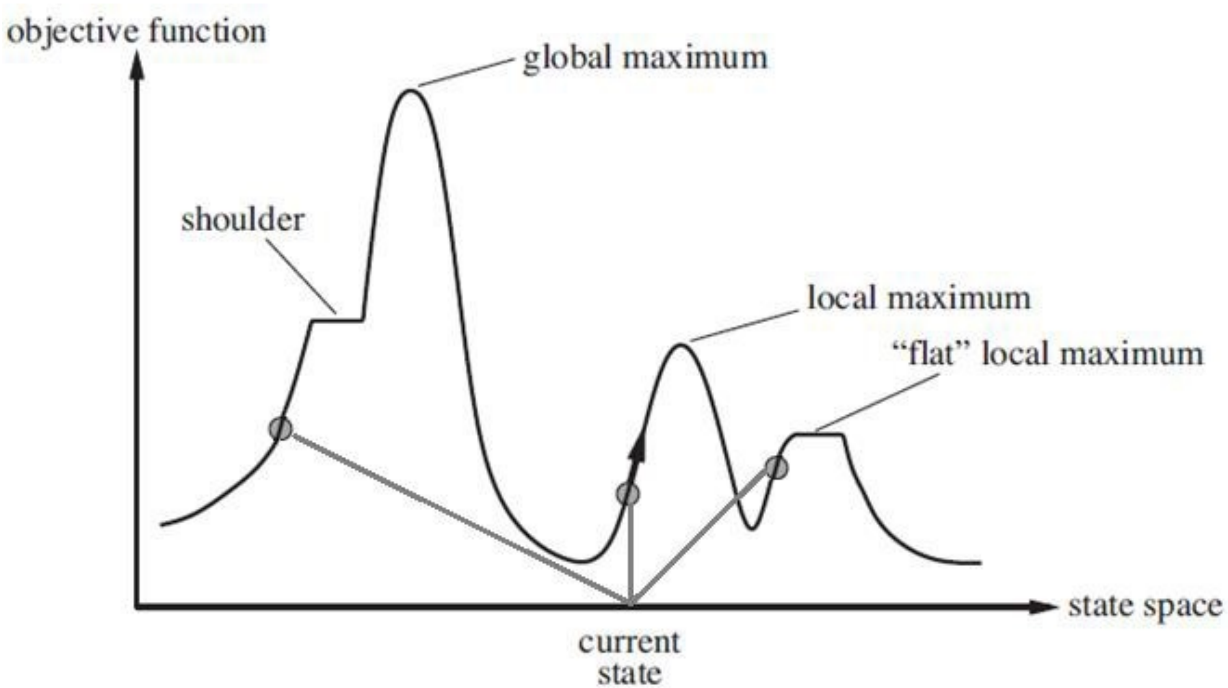
- State Space: assignment of booleans to facility locations and facility locations to cities.
- Sampling

1. Create random number p from {1,…,9} (select position to change)
2. Do
   - If $1 \leq p \leq 3$: switch boolean state
   - Else: Create random number from {1,…,3}\{A[p]} (select new location)

Local Beam Search

Local Beam Search Intuition

- Instead of following a single line through the state space, Local Beam Search follows multiple lines (a beam)



- In each step, we select the k best states from the set of all neighbors of all k current states

Local Beam Search

k_current ← select k random states
do
  neighbors ← all neighbors of all current states
  k_best_neighbors ← best k states from *neighbors*
  if no state from *k_best_neighbor* improves
  current value
    return *k_current*
  k_current← *k_best_neighbors*
until termination condition is met

- Termination conditions can be chosen as before

Local Beam Search vs Parallel Hill-Climbing

- **Parallel Hill-Climbing**: in each iteration, select best neighbor for each of the k states independently
- **Local Beam Search**: in each iteration, select the k best neighbors of all current states

Stochastic Beam Search

- Local Beam Search can concentrate on a small region of the search space too early
- Stochastic Beam Search alleviates this problem by using randomized selection similar to Stochastic Hill-Climbing

k_current ← select k random states
do
  neighbors ← all neighbors of all current states
  k_best_neighbors ← k 'random' states from *neighbors*
  if no neighbor improves current value
    return *k_current*
  k_current← *k_best_neighbors*
until termination condition is met

Random Selection

k_current ← select k random states
do
  neighbors ← all neighbors of all current states
  k_best_neighbors ← k 'random' states from *neighbors*
  if no neighbor improves current value
    return *best state from k_current*
  k_current← *k_best_neighbors*
until termination condition is met

- Again, random selection is usually not completely random
- probability of selecting a state should increase with its value
- In this way, we balance diversity (exploration) and intensification

Genetic Algorithms

Genetic Algorithms: Motivation

- Local Beam Search bears some resemblance to natural selection:
  - Neighbors (offspring) of states (organisms) populate next generation
  - Likelihood of survival depends on value (fitness)
- Genetic Algorithms extend this analogy
  - In each step, selected individuals reproduce
  - Selection probability increases with fitness (value)
  - There is a small probability of mutation
  - Offspring usually replaces other individuals (that die)

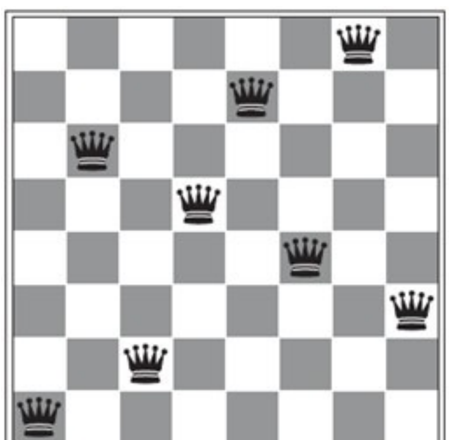
Apply Genetic Algorithms to a Problem

Steps for applying genetic algorithms to a problem:

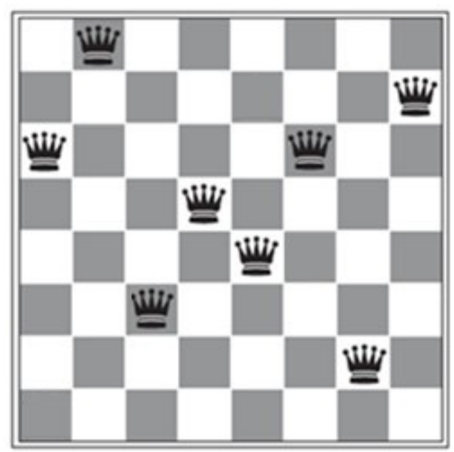
1. Define a suitable representation
2. Define an evaluation function (fitness function)
3. (Sometimes: Define special reproduction and mutation rules.)

Population (States)

- Genetic Algorithms work on a population of individuals
- Each individual is a state represented as a chromose (e.g. string) over a set of genes (e.g. set of characters)
- For 8-Queens problem, we could use string consisting of 8 digits from {1,…8}, where i-th digit is row position of i-th queen



83742516

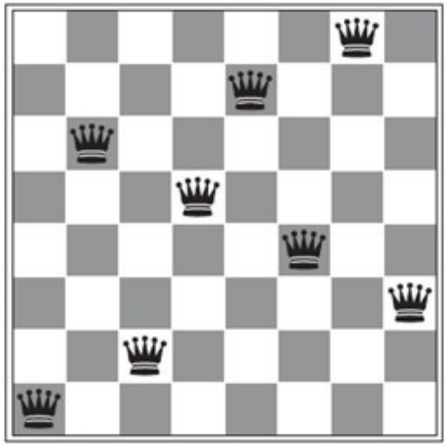


31645372

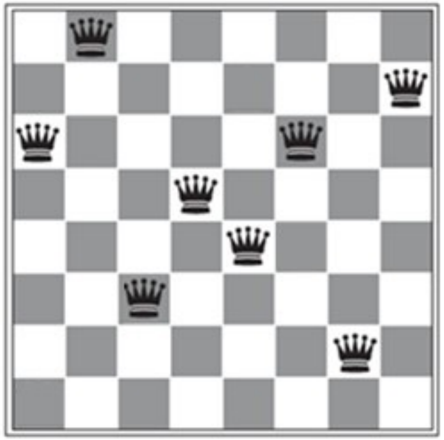
Fitness (Objective Function)

- Fitness corresponds to objective function
- For 8-Queens problem, we can reuse our old value function
- In order to get a non-negative fitness function, we could count the number of pairs of queens that cannot attack each other

(28 – number of attacks between pairs)



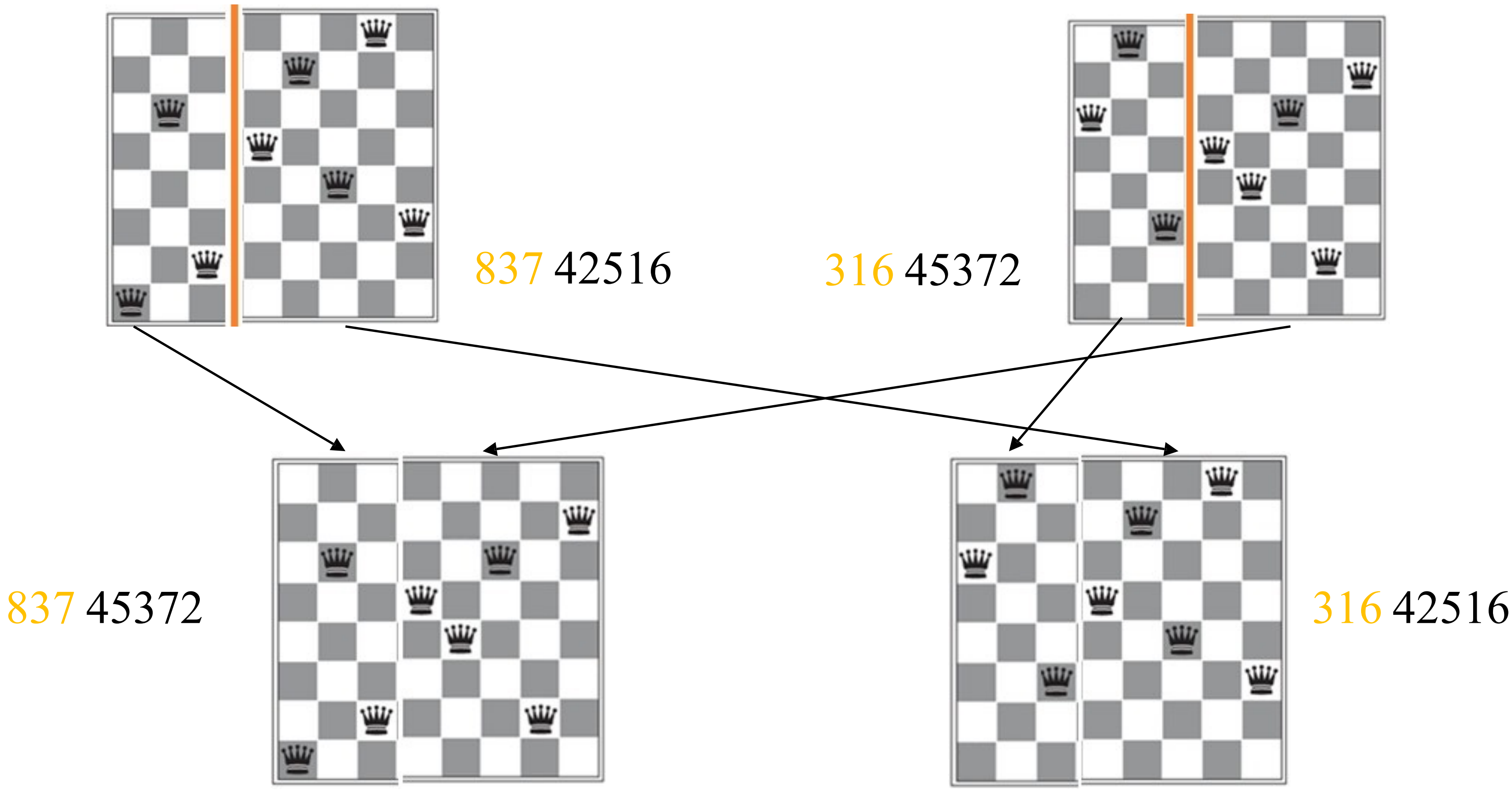
Fitness -1 or 27 (28–1)



Fitness -6 or 22 (28-6)

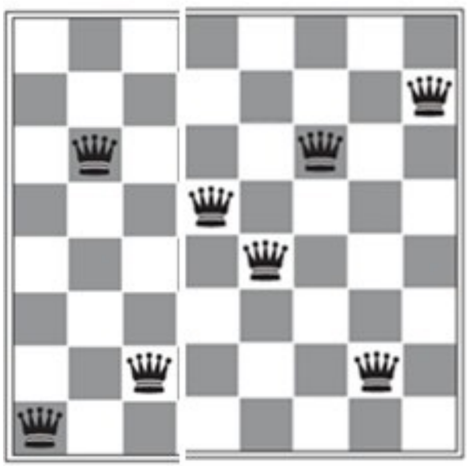
Reproduction (Recombination)

- Reproduction: choose a random crossover point in chromosome
- Create offspring by crossing parents at this point (1-point crossover)

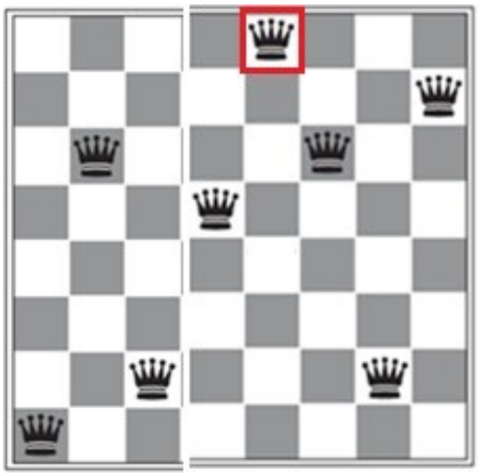


Mutation

- With small probability, mutation of offspring occurs
- E.g. replace one gene in chromosome randomly



837 45372



837 41372

Fitness-proportionate Selection

- We often want to select chromosomes based on their fitness

C_1: 10 | C_2: 30 | C_3: 20 | C_4: 25 | C_5: 15

Fitness-Proportionate (Reproduction)

C	Value	Probability
1	10	0.1
2	30	0.3
3	20	0.2
4	25	0.25
5	15	0.15
Sum	100	1

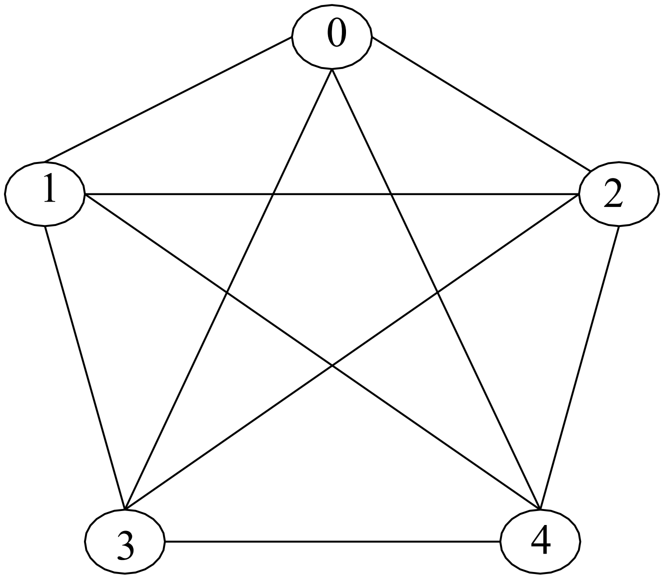
Fitness-Antiproportionate (Removal)

C	Value	Probability
1	1/10	0,34
2	1/30	0,12
3	1/20	0,17
4	1/25	0,14
5	1/15	0,23
Sum	0.29	1

A Simple Genetic Algorithm

population ← create k chromosomes at random
repeat
  for i = 1 **to** k **do**
    x ← select random chromosome based on fitness
    y ← select random chromosome based on fitness
    child ←reproduce(x , y)
    if (random() < 0.05)
      child ←mutate(child)
    add child to population
    remove random chromosome based on fitness
until some chromosome is fit enough, or time limit is reached
return the best chromosome in population

Example: Traveling Salesman Problem



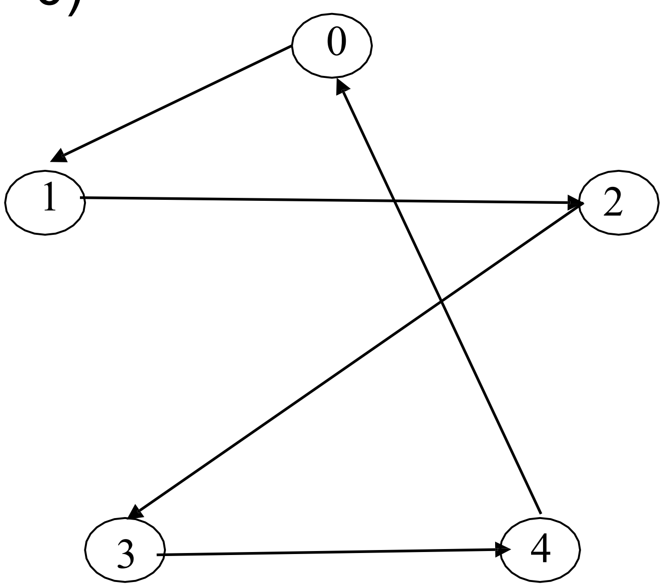
	0	1	2	3	4
0	0	3	4	5	6
1	3	0	6	2	1
2	4	6	0	3	4
3	5	2	3	0	7
4	6	1	4	7	0

(i-th column shows cost from node i to other nodes)

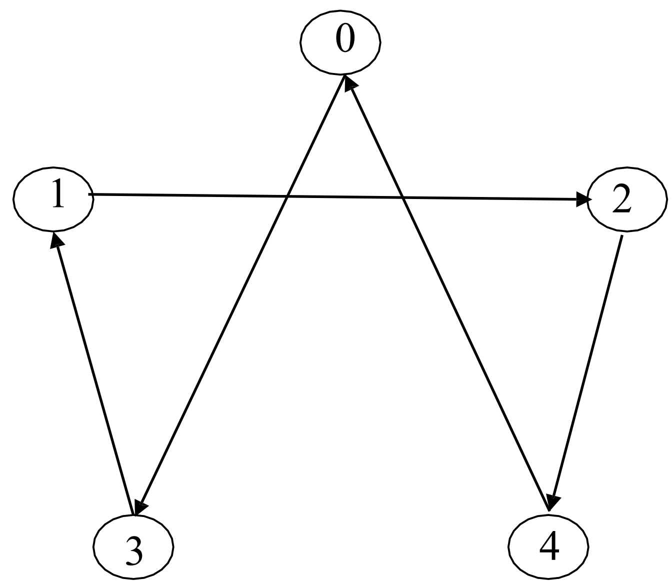
- TSP: find cyclic route that starts from node 0, visits each node exactly once and minimizes the overall edge cost
- Define
  - Genes
  - Chromosomes
  - Fitness (value) of individuals
  - Mutation operation
- Does crossover operation make sense?

Solution: Representation

- Genes {1, 2, 3, 4}
- Chromosomes are strings of length 4, where i-th gene (letter) represents the node that is visited at time i (first node is fixed to be 0)



1234

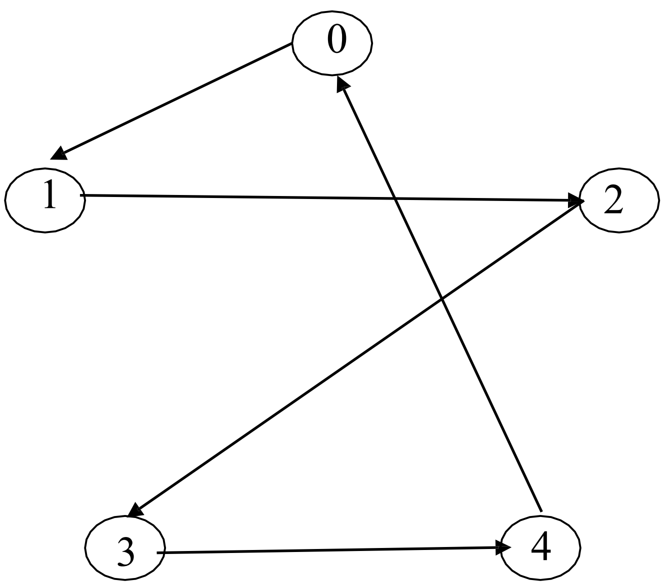


3124

Solution: Fitness

	0	1	2	3	4
0	0	3	4	5	6
1	3	0	6	2	1
2	4	6	0	3	4
3	5	2	3	0	7
4	6	1	4	7	0

- Fitness is 35 (rough upper bound on max. cost) minus the cost of getting from i-th to (i+1)-th node and from fithth node back to first node

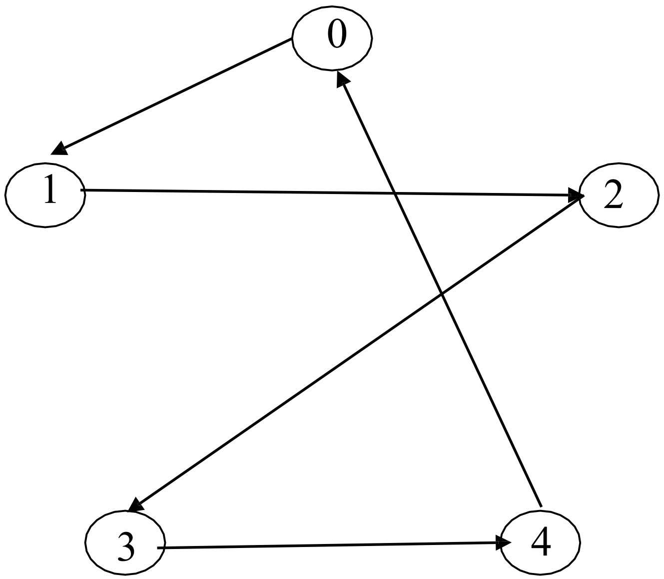


1234

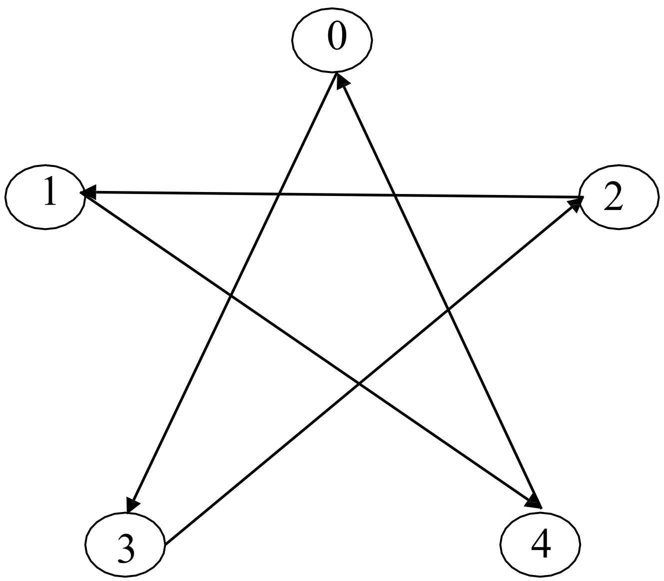
- Cost from 0 to 1: 3
- Cost from 1 to 2: 6
- Cost from 2 to 3: 3
- Cost from 3 to 4: 7
- <u>Cost from 4 to 0: 6</u>
- Overall cost: 25
- Fitness: 35-25 = 10

Solution: Mutation

- Mutation operation switches two genes at random
- E.g. switch first and third gene

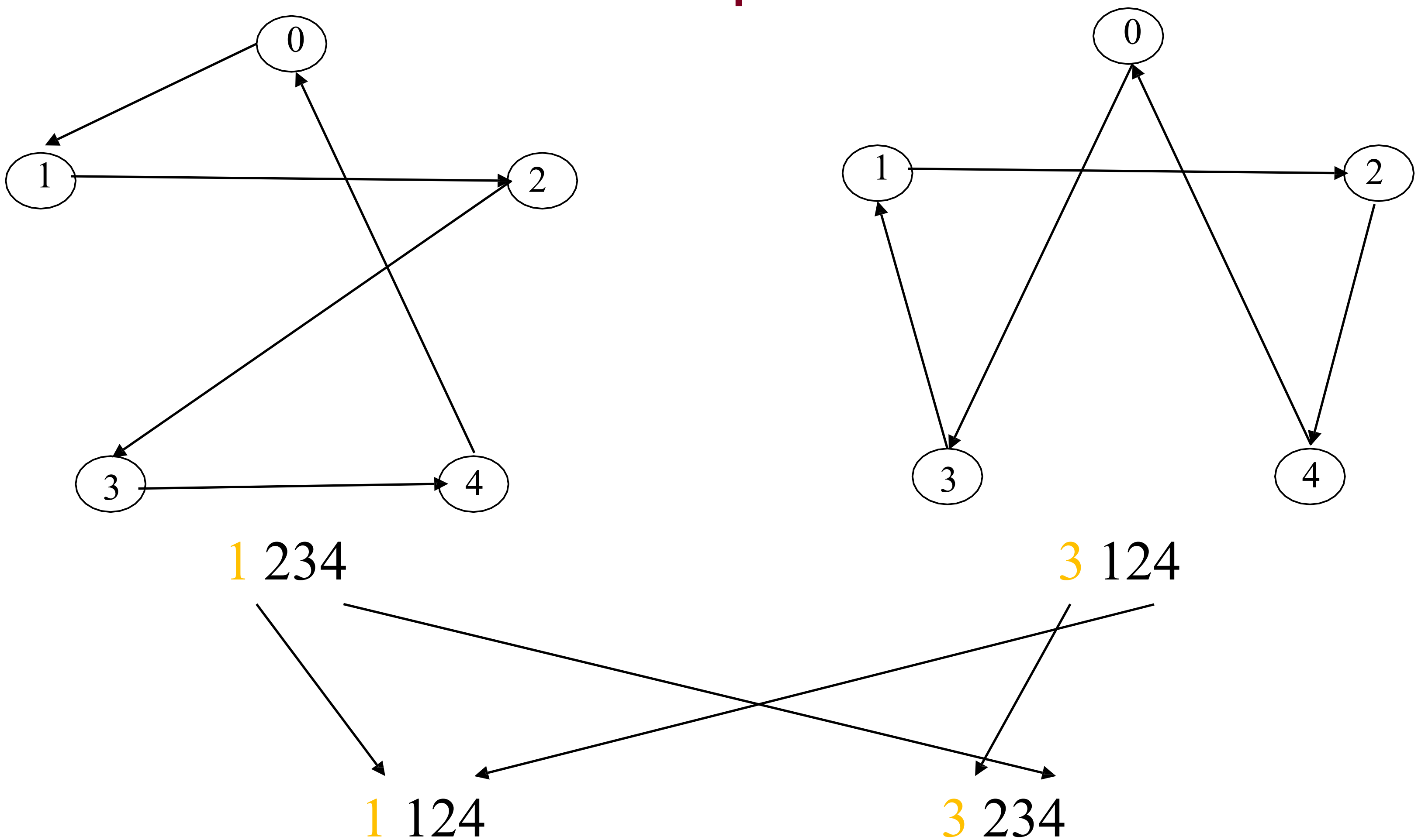


1234



3214

Solution: Problem with Reproduction



Naive reproduction yields invalid solutions

Reproduction of Permutations

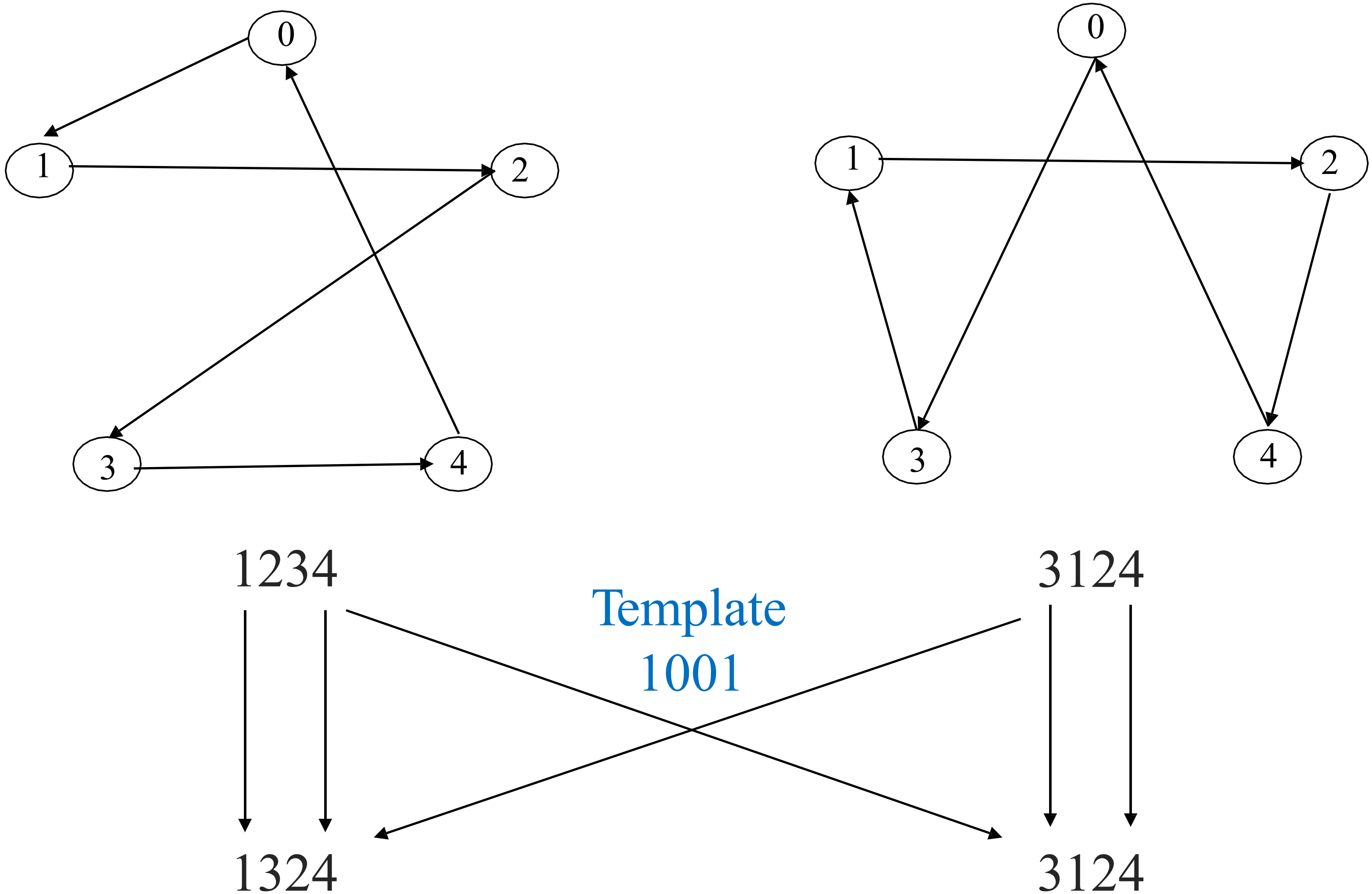
- If chromosomes correspond to permutations, naive crossover reproduction can yield invalid solutions
- the problem can be fixed by
  - Changing the representation
  - Designing an additional repair operation that is applied after recombination
  - Using special crossover techniques for permutations

Uniform-Order Crossover

- Uniform-Order Crossover
  - Select two parents at random
  - Create random binary template
  - Child 1 is created by using genes of parent 1 at 1-positions. Fill gaps with the remaining elements according to the order given by parent 2
  - Child 2 is created by switching the roles of parent 1 and parent 2

A	B	C	D	E	F	G	Parent P_1
E	B	D	C	F	G	A	Parent P_2

Solution: Problem with Reproduction



Variants

- Genetic Algorithms can be configured by different
  - Selection operators
  - Reproduction operators
  - Mutation operators

- Hybrid Genetic Algorithms combine genetic algorithms with other search techniques
  - Memetic Algorithms apply a (fast) local search algorithm to each newly created individual to make it locally optimal
  - Hill-Climbing is well suited for this purpose because it is fast

Further Readings

The presented slides are manily Dr. Tobias Thelen slides

Lecture is mainly based on:

Russell, S., Norvig, P. Artificial Intelligence - A modern approach. Pearson Education: 2010.

More details on presented (and similar algorithms) can be found in:

Burke, E. K., & Kendall, G. Search methodologies - Introductory Tutorials in Optimization and Decision Support Techniques. Springer US: 2014.