COMP 424 - Artificial Intelligence
Lecture 4: Search for optimization

Instructor: Jackie CK Cheung (jcheung@cs.mcgill.ca)

Questions

True or False (and explain your answer):

1. Both breadth-first search and A* are complete.
2. A heuristic can be inadmissible yet consistent.
3. A* is typically more space-efficient than iterative deepening.
4. Depth-first search and A* have the same time complexity.
5. Heuristic search is optimal for general step costs.
6. Depth-first search is a special case of greedy (best-first) search.

Overview

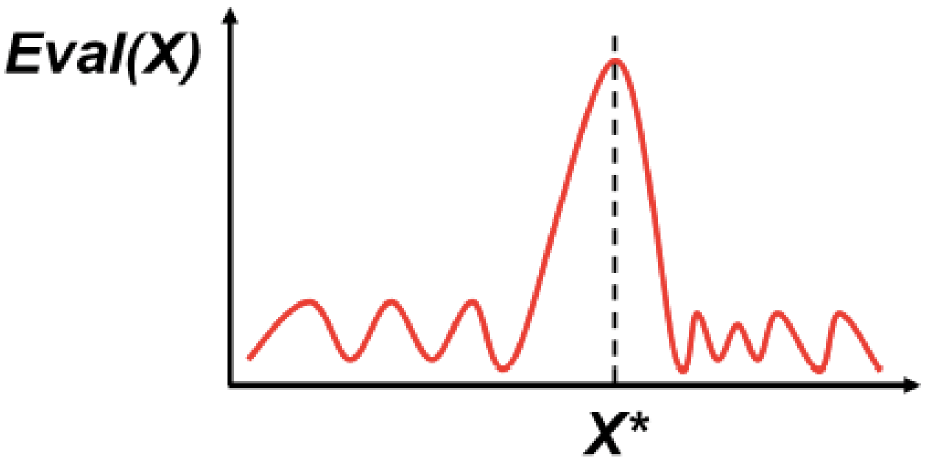
- Uninformed search:
  - Assumes **no knowledge** about the problem.
  - BFS, DFS, Iterative deepening
- Informed search:
  - Use **knowledge about the problem**, in the form of a heuristic.
  - Best-first search, heuristic search, A* (and extensions)
- Search for optimization problems:
  - Search over large-dimensional (continuous) spaces.
  - Iterative improvement algorithms:
    1. hill climbing
    2. simulated annealing

Today!

Optimization problems

Typically characterized by:

- Large (continuous, combinatorial) state space, ***X***.
- Searching all possible solutions is infeasible.
- A (non-uniform) cost function, which we want to optimize.



Optimization problems

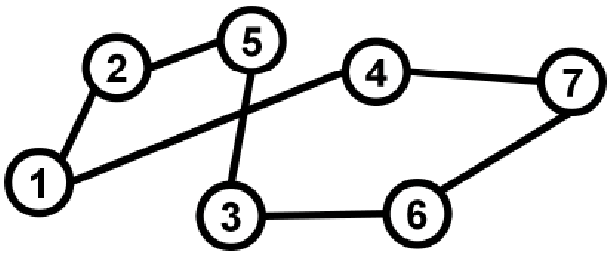
Typically characterized by:

- Large (continuous, combinatorial) state space, ***X***.
- Searching all possible solutions is infeasible.
- A (non-uniform) cost function, which we want to optimize.
- We are satisfied to achieve a "good" solution.
- In some cases, constraints have to be satisfied.

Mathematical optimization is a field of its own. Here, we focus on those problems that arise most frequently in AI.

Traveling salesman problem (TSP)

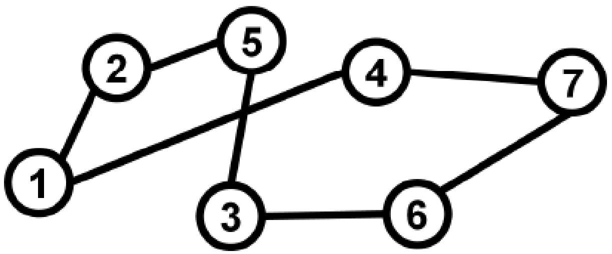
- **Example**:
  - Given a set of vertices and distance between pairs.
  - Goal: construct the shortest path that visits every vertex exactly once.
  - A path that satisfies the goal is called a tour. ***X_1*** (above) is a tour, but not an optimal one.



X_1 = {1 2 5 3 6 7 4}

Example: Traveling salesman problem (TSP)

- **Example**:
  - Given a set of vertices and distance between pairs.
  - Goal: construct the shortest path that visits every vertex exactly once.
  - A path that satisfies the goal is called a tour. ***X_1*** (above) is a tour, but not an optimal one.



$X_1 = \{1\ 2\ 5\ 3\ 6\ 7\ 4\}$

- Often easy to find *some solution* to the problem.
- But provably very hard (NP-complete) to find the ***best*** solution. But we still want a ***good*** solution!

Many more examples

- Scheduling
  - Given: a set of tasks to be completed, with durations and mutual constraints (e.g. task ordering, joint resources).
  - Goal: generate shortest schedule (assignment of start times to tasks.)

Many more examples

- Scheduling
  - Given: a set of tasks to be completed, with durations and mutual constraints (e.g. task ordering, joint resources).
  - Goal: generate shortest schedule (assignment of start times to tasks.)
- Digital circuit layout
  - Given: a board, components and connections.
  - Goal: place each component on the board such as to maximize energy efficiency, minimize production cost, …

Many more examples

- Scheduling
  - Given: a set of tasks to be completed, with durations and mutual constraints (e.g. task ordering, joint resources).
  - Goal: generate shortest schedule (assignment of start times to tasks.)
- Digital circuit layout
  - Given: a board, components and connections.
  - Goal: place each component on the board such as to maximize energy efficiency, minimize production cost, …
- User customization
  - Given: customers described by characteristics (age, gender, location, etc.) and previous purchases.
  - Goal: find a function from characteristics to products that maximizes expected gain.

Optimization problems are everywhere!

Characteristics

- An optimization problem is described by a set of states (=configurations) and an evaluation function.
- For interesting optimization problems, the *state space is too big* to enumerate all states, or the *evaluation function is too expensive to compute* for all states.

Characteristics

- An optimization problem is described by a set of states (=configurations) and an evaluation function.
- For interesting optimization problems, the *state space is too big* to enumerate all states, or the *evaluation function is too expensive to compute* for all states.

- A **state** is a **candidate solution**, not a description of the world.
- The state can be a partial or incorrect solution.
- The evaluation function corresponds to the path cost.

  E.g. in TSP, a tour is a state, and the length of the tour is the function (to be minimized).

Types of search for optimization problems

1. **Constructive methods**: Start from scratch and build up a solution.
   - This is the type of method we have seen so far.

2. **Iterative improvement/repair methods**: Start with a solution (which may be broken / suboptimal) and improve it.

Types of search for optimization problems

1. **Constructive methods**: Start from scratch and build up a solution.
   - This is the type of method we have seen so far.
   - e.g., **TSP:** start at the start city, add cities to form a complete tour
2. **Iterative improvement/repair methods**: Start with a solution (which may be broken / suboptimal) and improve it.
   - e.g., **TSP:** start with a complete tour and keep swapping cities to improve cost.

Types of search for optimization problems

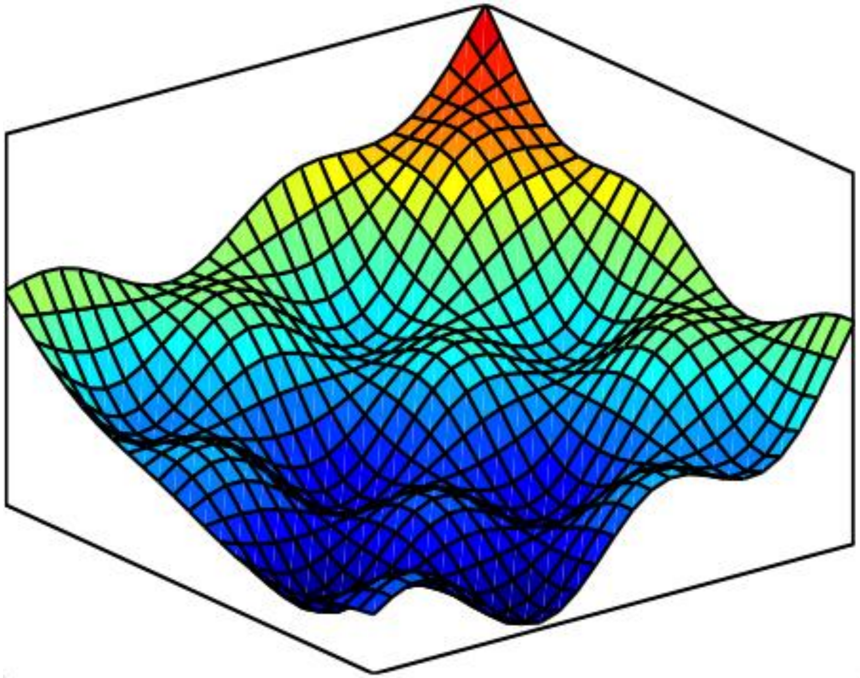
1. **Constructive methods**: Start from scratch and build up a solution.
   - This is the type of method we have seen so far.
   - e.g., **TSP:** start at the start city, add cities to form a complete tour
2. **Iterative improvement/repair methods**: Start with a solution (which may be broken / suboptimal) and improve it.
   - e.g., **TSP:** start with a complete tour and keep swapping cities to improve cost.

- In both cases, the search is **local**:
  - Consider one solution, apply modification to generate the next one.
  - Only consider a solution at a time, don't memorize previous solutions explored.

Iterative improvement algorithm

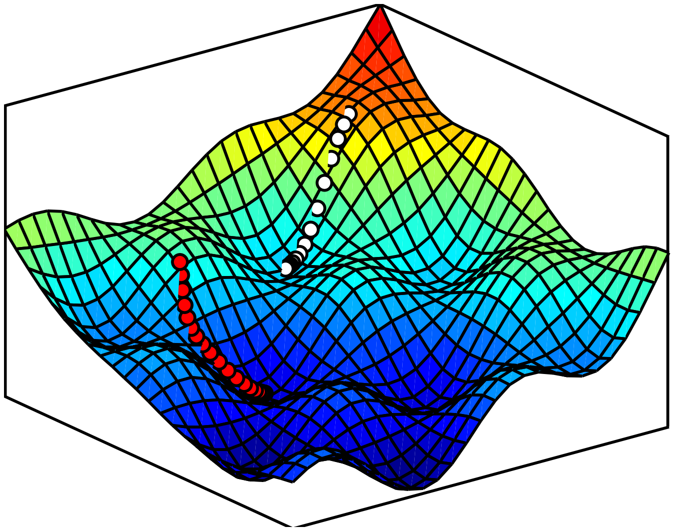
- **Intuition**: Consider all possible solutions laid out on a landscape. We want to find the highest (or lowest) point.

- This landscape is often high-dimensional.



A generic local search algorithm

- Start from an initial configuration X_0.
- Repeat until satisfied:
  - Generate the set of neighbours of X_i and evaluate them.
  - Select one of the neighbours, X_{i+1}.
  - The selected neighbor becomes the current configuration.



A generic local search algorithm

- Start from an initial configuration X_0.
- Repeat until satisfied:
  - Generate the set of neighbours of X_i and evaluate them.
  - Select one of the neighbours, X_{i+1}.
  - The selected neighbor becomes the current configuration.
- Important questions:
  - How do we choose the set of neighbours to consider?
  - How do we select one of the neighbours?
- Defining the set of neighbours is a *design choice* (like choosing the heuristic for A*) and has crucial impact on performance.

What moves should we consider?

- Case 1: Robot planning
  - Start with initial state = random position.
  - Move to an adjacent position.
  - Terminate when goal is reached.

What moves should we consider?

- Case 1: Robot planning
  - Start with initial state = random position.
  - Move to an adjacent position.
  - Terminate when goal is reached.

- Case 2: Traveling Salesman Problem
  - Start with initial state = a random (possibly incomplete/illegal) tour.
  - Swap cities to obtain a new partial tour.
  - Terminate when constraints are met.

Hill-climbing (aka greedy local search, gradient ascent/descent)

- Start from an initial configuration X_0 with value $E(X_0)$.
- Repeat until satisfied:
  - Generate the set of neighbours of X_i and their value $E(X_i)$.
  - Let $E_{max} = max_i\ E(X_i)$ be the value of the **best** successor, and $i^* = argmax_i\ E(X_i)$ be the index of the **best** successor.
  - If $E_{max} \leq E$, return X (we are at an optimum).
  - Else let $X \leftarrow X_{i^*}$, and $E = E_{max}$.

Properties of hill-climbing

- Variant of best-first search. Very popular in AI.
  - Trivial to program!
  - Requires no memory of where we've been (no backtracking).
  - Can handle very large problems.

Properties of hill-climbing

- Variant of best-first search. Very popular in AI.
  - Trivial to program!
  - Requires no memory of where we’ve been (no backtracking).
  - Can handle very large problems.

- Important to have a “good” set of neighbours.
  - Small neighbourhood = fewer neighbours to evaluate, but possibly worse solution.
  - Large neighbourhood = more computation, but maybe fewer local optima, so better final result.

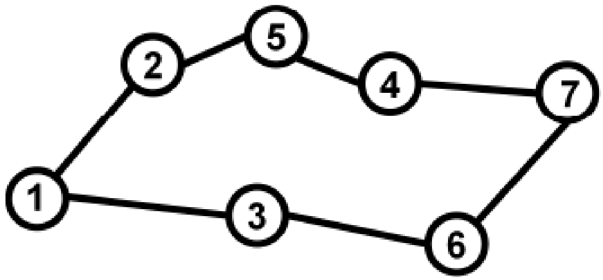
Local vs Global Optimum

- **Global optimum** = The optimal point over the **full space** of possible configurations.
- **Local optimum** = The optimal point over the **set of neighbours**. One of the (possibly many) optimums.
- Important distinction (throughout the course) about algorithms that are globally vs locally optimal.

$S = \{1,\ldots,100\}$

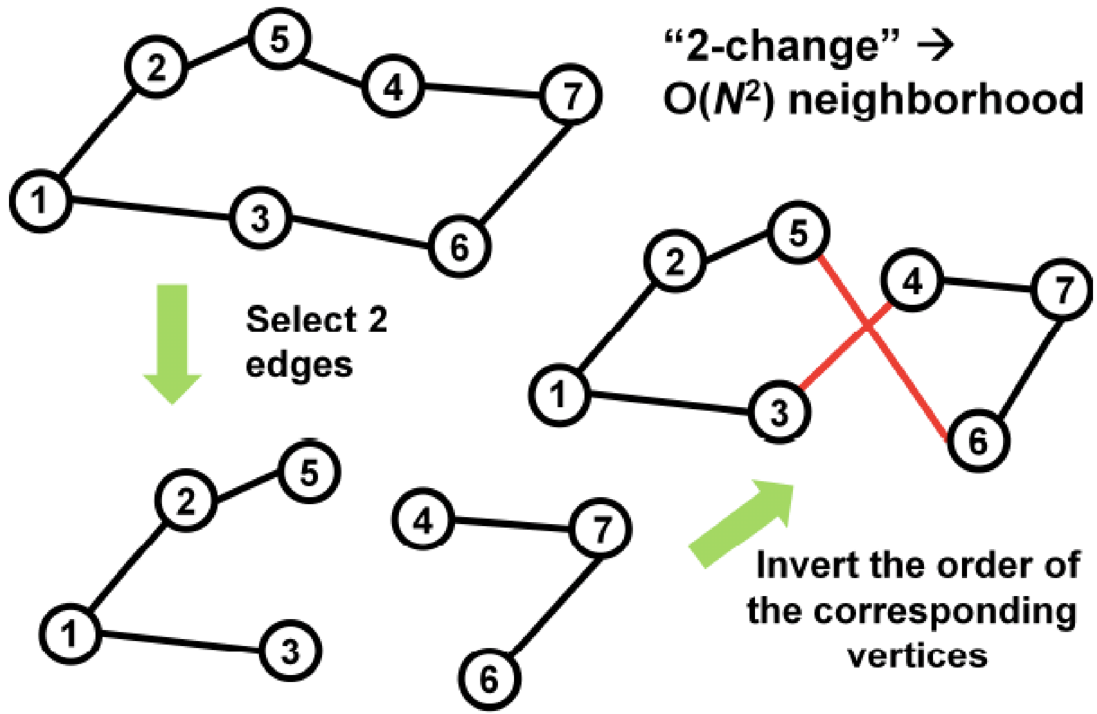
$Neighbors(X) = \{X\text{-}1, X\text{+}1\}$

Example: TSP



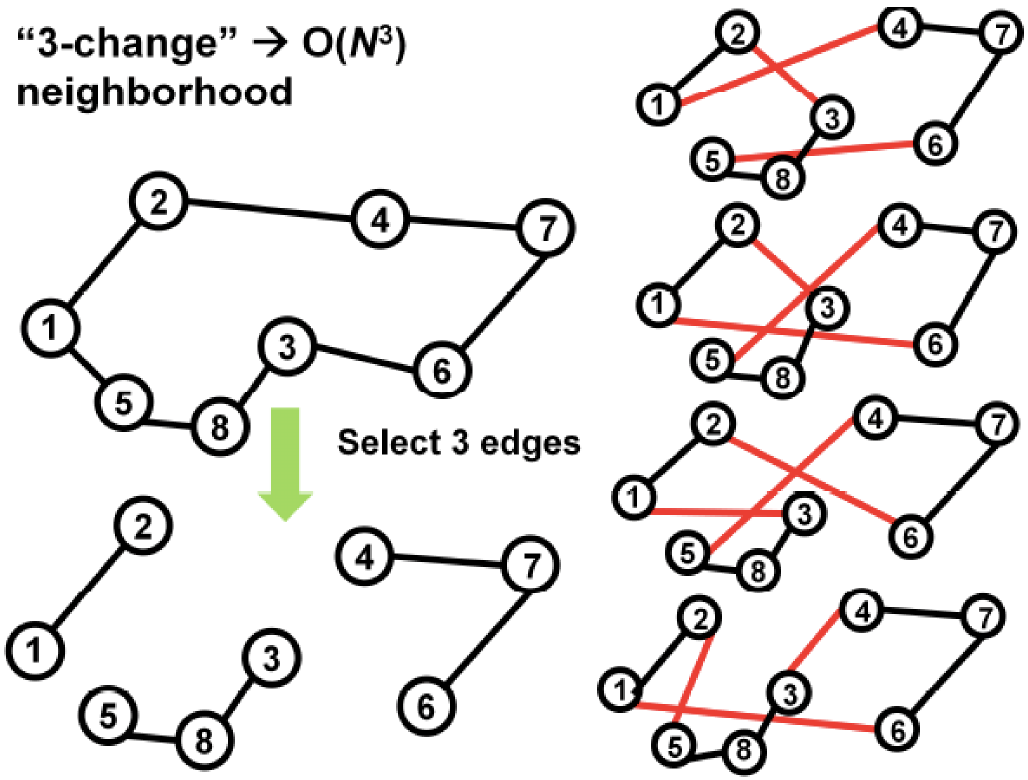
- What neighbours should we consider?
- How many neighbours is that?

Example: TSP swapping 2 nodes



$O(n^2)$ comes from the fact that we have n edges in a tour, and choose two of them to swap, so there are $\binom{n}{2}$ possible next tours

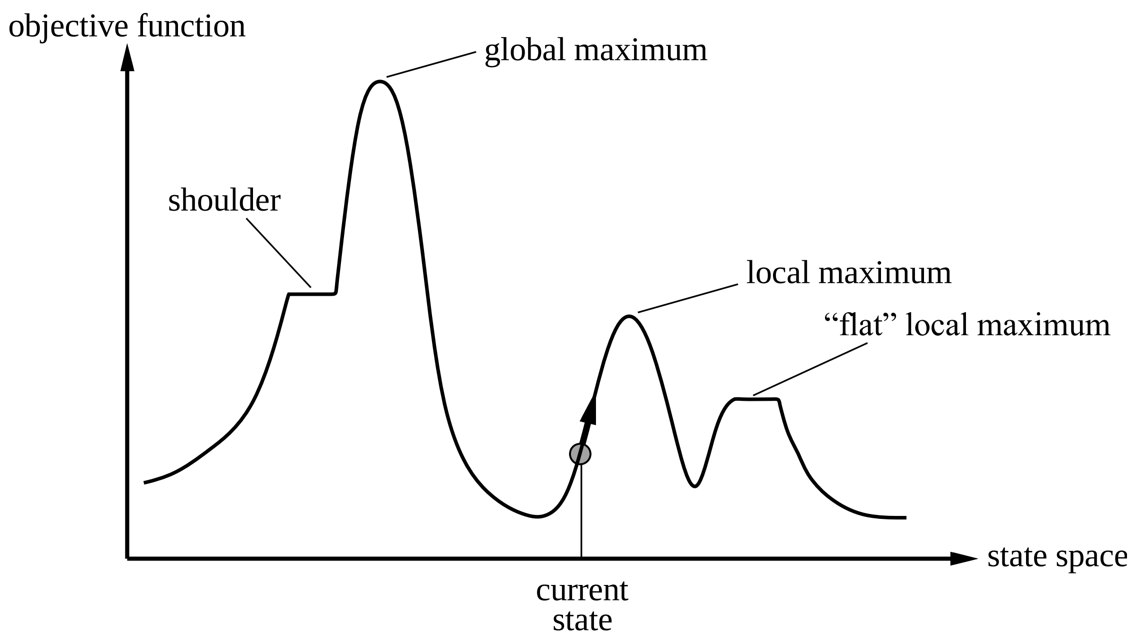
Example: TSP swapping 3 nodes



There are $\binom{n}{3}$ combinations of edges to choose, and for each set of edges, more than one possible neighbor

Problems with hill climbing

- Can get stuck in a local maximum or in a plateau.



- Relies heavily on having a good evaluation.

Improvements to hill climbing

- Quick fix:
  - When stuck in a plateau or local maximum, use random re-starts.

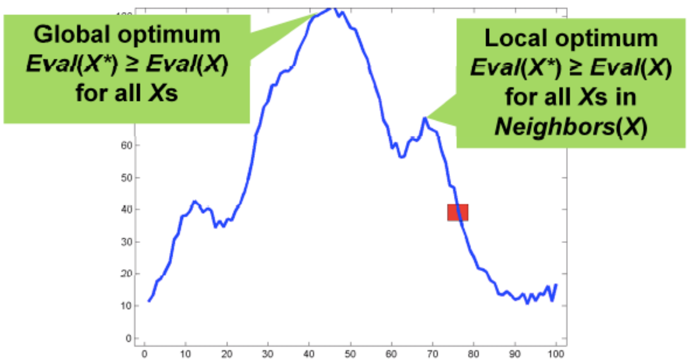
- Slightly better fix:
  - Instead of picking the next best move, **pick any move that produces an improvement**. (Called randomized hill climbing.)

Improvements to hill climbing

- Quick fix:
  - When stuck in a plateau or local maximum, use random re-starts.

- Slightly better fix:
  - Instead of picking the next best move, **pick any move that produces an improvement**. (Called randomized hill climbing.)

- But sometimes we need to pick apparently worse moves to eventually reach a better state.



S = {1,…,100}

$Neighbors(X)$ = {X-1,X+1}

Simulated annealing

Similar to hill climbing, but:

- allows some “bad moves” to try to escape local maxima.
- decrease size and frequency of “bad moves” over time.

Simulated annealing

Similar to hill climbing, but:

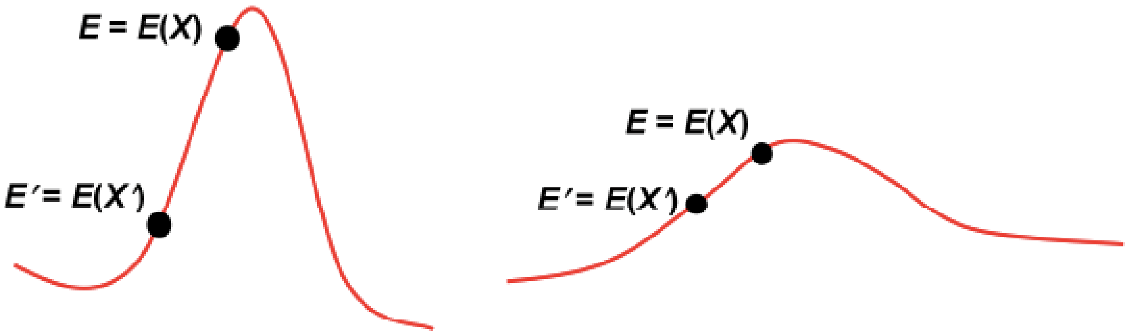
- allows some "bad moves" to try to escape local maxima.
- decrease size and frequency of "bad moves" over time.

Algorithm:

- Start from an initial configuration X_0 with value $E(X_0)$.
- Repeat until satisfied:
  - Let X_i be a random neighbour of X with value $E(X_i)$.
  - If $E_i > E_{max}$, let $X_{i^*} \leftarrow X_i$ and let $E_{max} = E$ (we found a new better sol'n).
  - If $E_i > E$ then $X \leftarrow X_i$ and $E \leftarrow E_i$.
  - Else, with some probability p, still accept the move: $X \leftarrow X_i$ and $E \leftarrow E_i$.
- Return X_{i^*} .

What value should we use for *p*?

- Many possible choices:
  - A given fixed value.
  - A value that decays to 0 over time.
  - A value that decays to 0, and gives similar chance to “similarly bad” moves.
  - A value that depends on on how much worse the bad move is.



What value should we use for *p*?

- If the new value E_i is better than the old value E, move to X_i.
- If the new value is worse ($E_i<E$) then move to the neighboring solution with probability: $p = e^{-(E-Ei)/T}$ [Boltzmann distribution]

What value should we use for *p*?

- If the new value E_i is better than the old value E, move to X_i.

- If the new value is worse ($E_i<E$) then move to the neighboring solution with probability: $p = e^{-(E-Ei)/T}$ [Boltzmann distribution]
  - $T>0$ is a parameter called the temperature, which typically starts high, then decreases over time towards 0.
  - If T is very close to 0, the probability of moving to a worse solution is almost 0.
  - We can gradually decrease T by multiplying by constant $\alpha<0$ at every iteration.

Where does the Boltzmann distribution come from?

- For a solid, at temperature T, the probability of moving between two states of energy difference ΔE is $e^{-\Delta E / kT}$.
- If temperature decreases slowly, it will reach an equilibrium at which the probability of being in a state of energy E is proportional to $e^{-E / kT}$.
- So states of low energy (relative to T) are more likely.
- In our case, states with better value will be more likely.

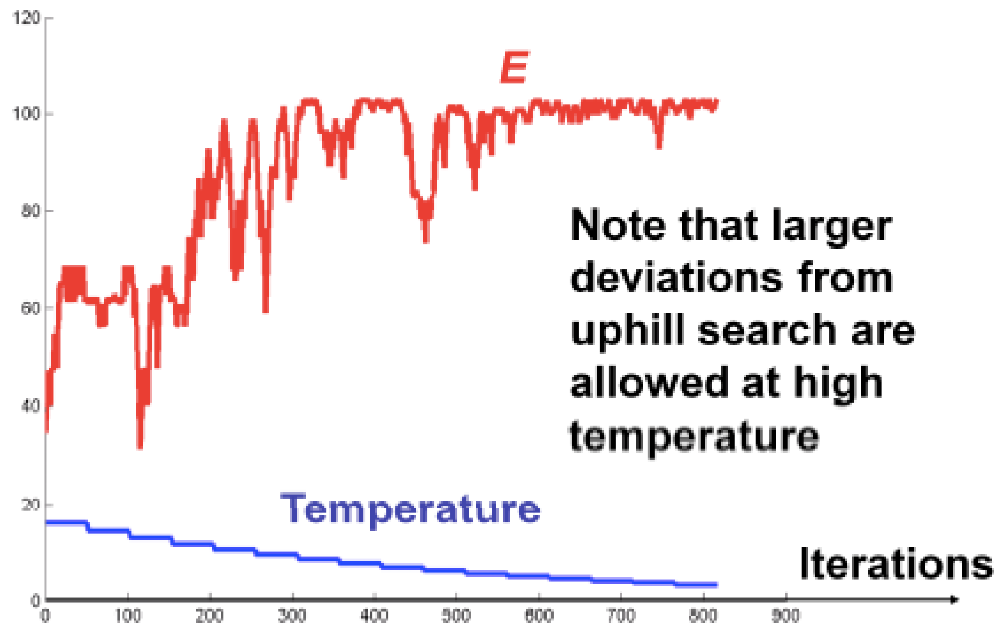
Properties of simulated annealing

- What happens when T is high?
  - Algorithm is in an **exploratory phase** (even bad moves have a high chance of being picked).
- What happens when T is low?
  - Algorithm is in an **exploitation phase** (the “bad” moves have very low probability).

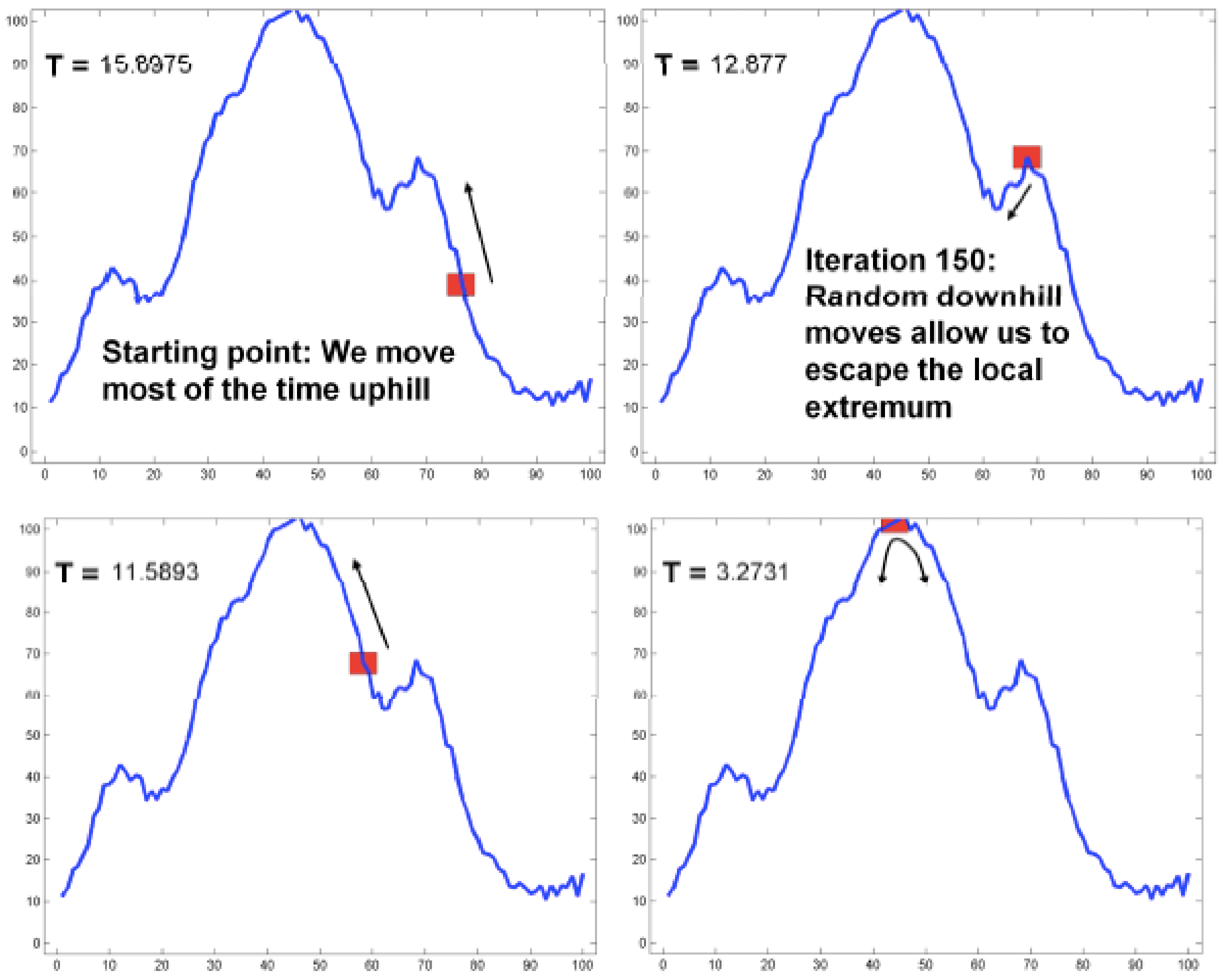
Properties of simulated annealing

- If *T* decreases slowly enough, simulated annealing is guaranteed to reach the **optimal solution** (i.e., find the global maximum).

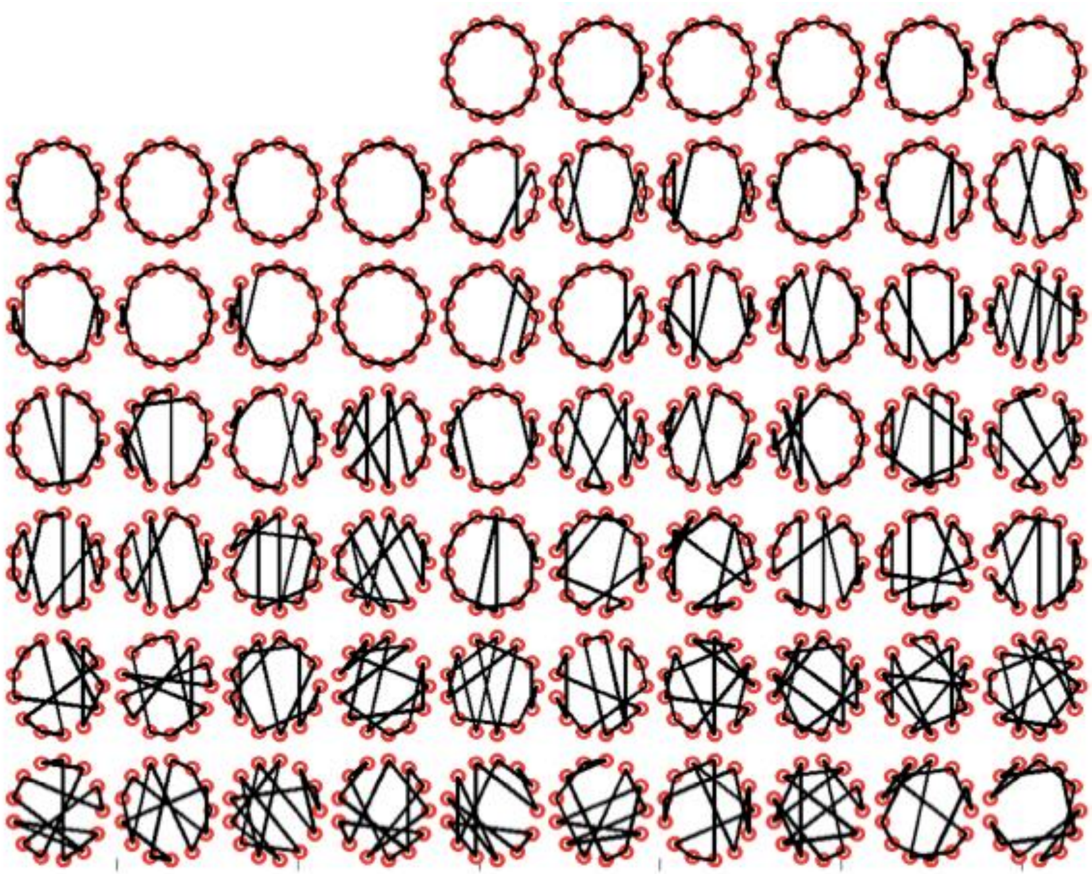
- But it may take an infinite number of moves!



Example

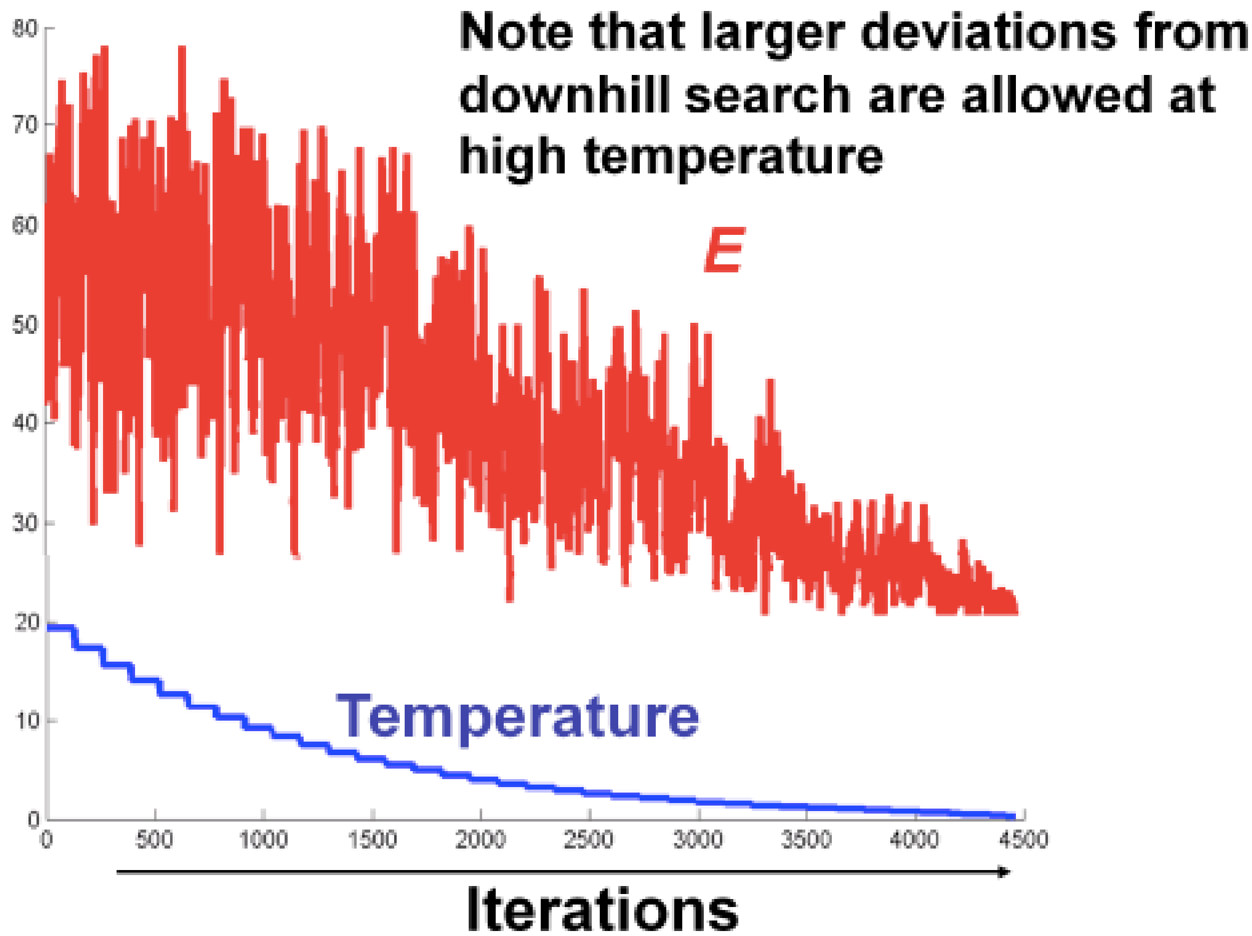

T = 15.8975
Starting point: We move most of the time uphill
T = 12.877
Iteration 150: Random downhill moves allow us to escape the local extremum
T = 11.5893
T = 3.2731

TSP example: Searching configurations



The initial configuration is bottom right, final one is top left

TSP Example: Energy



Question

- Under what conditions does simulated annealing perform better than hill-climbing?

- Would you ever prefer hill-climbing? If so, when?

Simulated annealing in practice

- Very useful algorithm, used to solve hard optimization problems.
  - E.g. Protein design, scheduling large transportation fleets.
- The temperature annealing schedule is crucial (design choice!)
  - Cool too fast: converge to sub-optimal solution.
  - Cool too slow: don't converge.

Simulated annealing in practice

- Very useful algorithm, used to solve hard optimization problems.
  - E.g. Protein design, scheduling large transportation fleets.
- The temperature annealing schedule is crucial (design choice!)
  - Cool too fast: converge to sub-optimal solution.
  - Cool too slow: don't converge.

- Simulated annealing is an example of a randomized search or **Monte-Carlo search**.
  - Basic idea: run around through the environment and explore it, instead of systematically sweeping. Very powerful idea!

Parallel search

- Run many separate searches (hill-climbing or simulated annealing) in parallel.
- Keep the best solution found.
- Search speed can be greatly improved by using many processors (including, most recently, GPUs).
- Especially useful when actions have non-deterministic outcomes (many possible successor states).

Summary

- Optimization problems are widespread and important.
- It is unfeasible to enumerate lots of solutions.
- Goal is to get a reasonable (not necessarily optimal) solution.

- Apply a local search and move in a promising direction.
  - Hill climbing (a.k.a. gradient ascent/descent) always moves in the (locally) best direction.
  - Simulated annealing allows some moves towards worse solutions.

- Search for optimization is a large field, with many variants on the algorithms described today.

Search for optimization problems:

- **Constructive methods**: Start from scratch and build up a solution.
  - Informed / uninformed methods.
- **Iterative improvement/repair methods**: Start with a solution (which may be broken / suboptimal) and improve it.
  - Hill-climbing, simulated annealing.
- **Global search**: Start from multiple states that are far apart, and go all around the state space.

Evolutionary computing

- Refers generally to computational procedures patterned after biological evolution.
- Many solutions (individuals) exist in parallel.
- Nature looks for the best individual (i.e. the fittest).

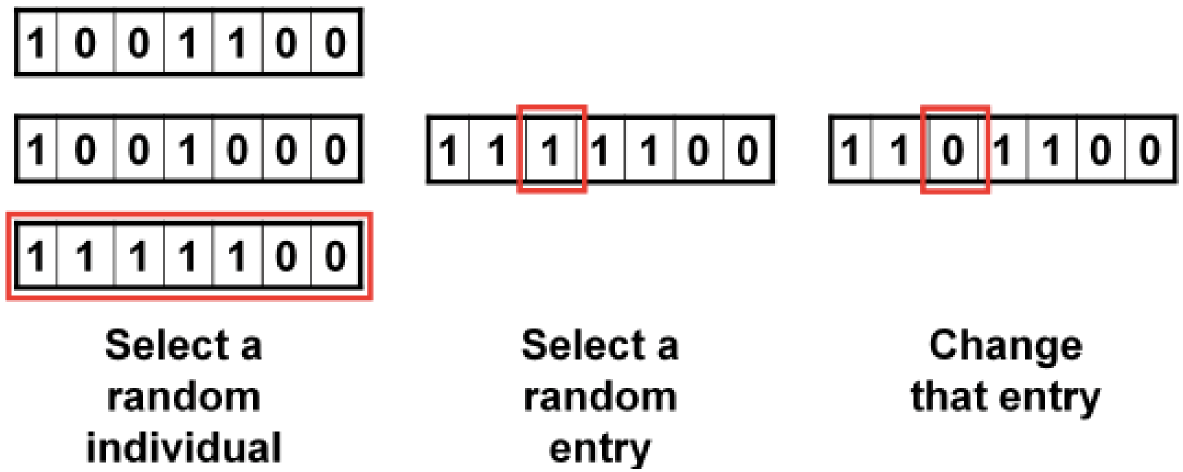
- Evolutionary search procedures are also parallel, perturbing probabilistically several potential solutions.

Genetic algorithms

- A candidate solution is called an individual.
  - In a traveling salesman problem, an individual is a tour
- Each individual has a fitness
  - fitness = numerical value proportional to quality of that solution
- A set of individuals is called a population.
- Populations change over generations, by applying operations to individuals.
  - operations = {mutation, crossover, selection}
- Individuals with higher fitness are more likely to survive & reproduce.
- Individual typically represented by a binary string:
  - allows operations to be carried out easily.

Mutation

- A way to generate desirable features that are not present in the original population by injecting random change.
  - Typically mutation just means changing a 0 to a 1 (and vice versa).
- The mutation rate controls prob. of mutation occurring
- We can allow mutation in all individuals, or just in the offspring.



Crossover

- Combine parts of individuals to create new individuals.
- Single-point crossover:
  - Choose a crossover point, cut individuals there, swap the pieces.

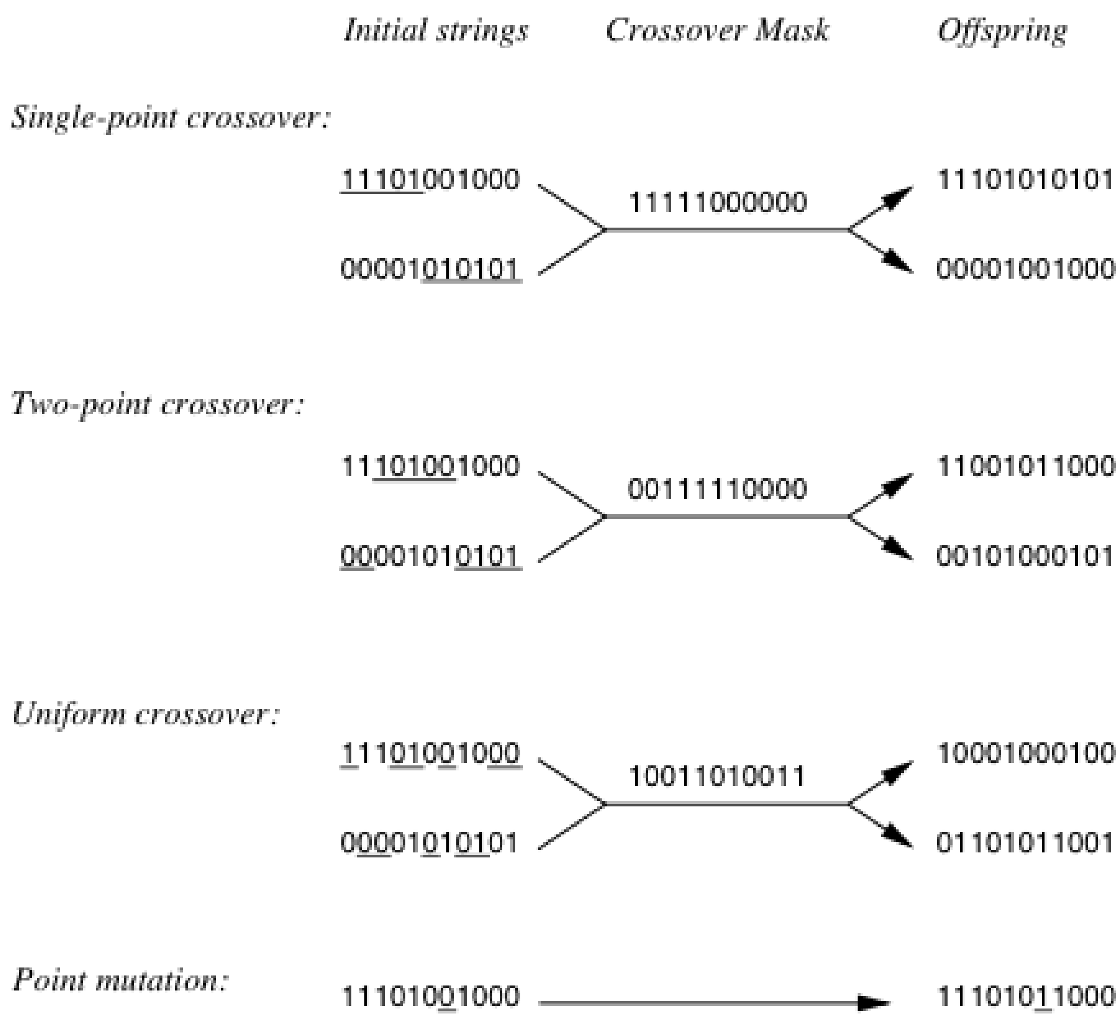
    E.g. **101|0101** → **101|1110**

    011|1110 → **011|0101**
- Implementation:
  - Use a crossover mask, which is a binary string

    E.g. mask = 1110000

- Multi-point crossover can be implemented with arbitrary mask.

Encoding operators as binary masks



Typical genetic algorithm

GA(Fitness, threshold, p, r, m)

- Initialize: $P \leftarrow p$ random individuals
- Evaluate: for each $h \in P$, compute *Fitness(h)*
- While max_h *Fitness(h) < threshold*
  - Select: Probabilistically select $(1-r)p$ members of P to include in P_s
  - Crossover: Probabilistically select $rp/2$ pairs of individuals from P.

    For each pair (h_1, h_2), produce two offspring by applying a crossover operator. Include all offspring in P_s.
  - Mutate: Invert a randomly selected bit in $m*p$ randomly selected members of P_s
  - Update: $P \leftarrow P_s$
  - Evaluate: for each $h \in P$, compute *Fitness(h)*
- Return the individual from P that has the highest fitness.

Selection: Survival of the fittest

- As in natural evolution, fittest individuals are more likely to survive.
- Several ways to implement this idea:
  1. Fitness proportionate selection:

     $$P(i) = \frac{Fitness(i)}{\sum_{j=1}^{p} Fitness(j)}$$

     Can lead to crowding (multiple copies being propagated).

  2. Tournament selection:

     Pick *i, j* at random with uniform probability. With prob *p* select the fitter one. Only requires comparing two individuals.

  3. Rank selection:

     Sort all hypothesis by fitness. Probability of selection is proportional to rank.

  4. Softmax (Boltzman) selection:

     $$P(i) = \frac{e^{Fitness(i)/T}}{\sum_{j=1}^{p} e^{Fitness(j)/T}}$$

Elitism

- The best solution can "die" during evolution
- In order to prevent this, the best solution ever encountered can always be "preserved" on the side
- If the "genes" from the best solution should always be present in the population, it can also be copied in the next generation automatically, bypassing the selection process.
- **Note that the best solution ever encountered is typically saved in hill climbing and simulated annealing as well.**

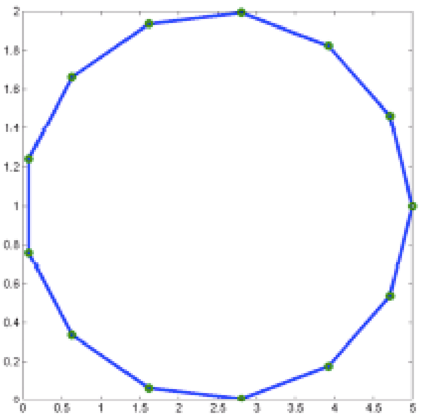
Genetic algorithms as search

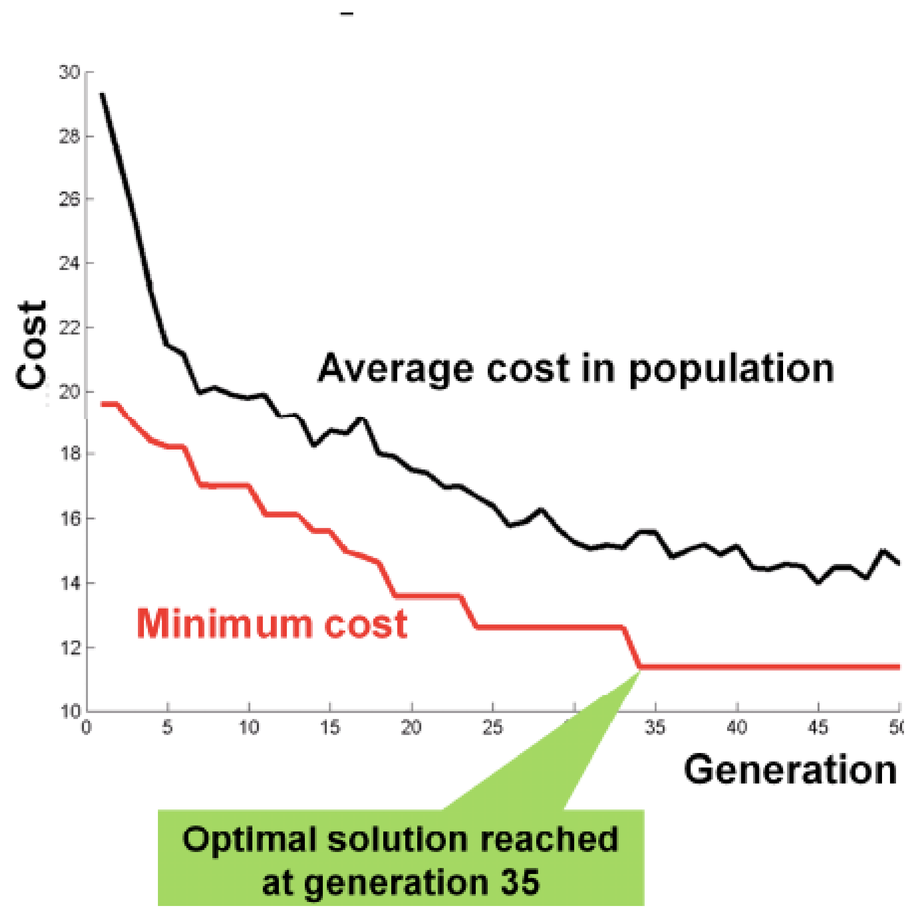
- **States**: possible solutions
- **Search operators**: mutation, crossover, selection
- Relation to previous search algorithms:
  - Parallel search, since several solutions are maintained in parallel
  - Hill-climbing on the fitness function, but without following the gradient
  - Mutation and crossover should allow us to get out of local minima
  - Very related to simulated annealing.

Example: Solving TSP with a GA

- Each individual is a tour.
- Mutation swaps a pair of edges (many other operations are possible and have been tried in literature.)
- Crossover cuts the parents in two and swaps them. Reject any invalid offsprings.
- Fitness is the length of the tour.
- Note that GA operations (crossover and mutation) described here are fancier that the simple binary examples given before.

Example: Solving TSP with a GA



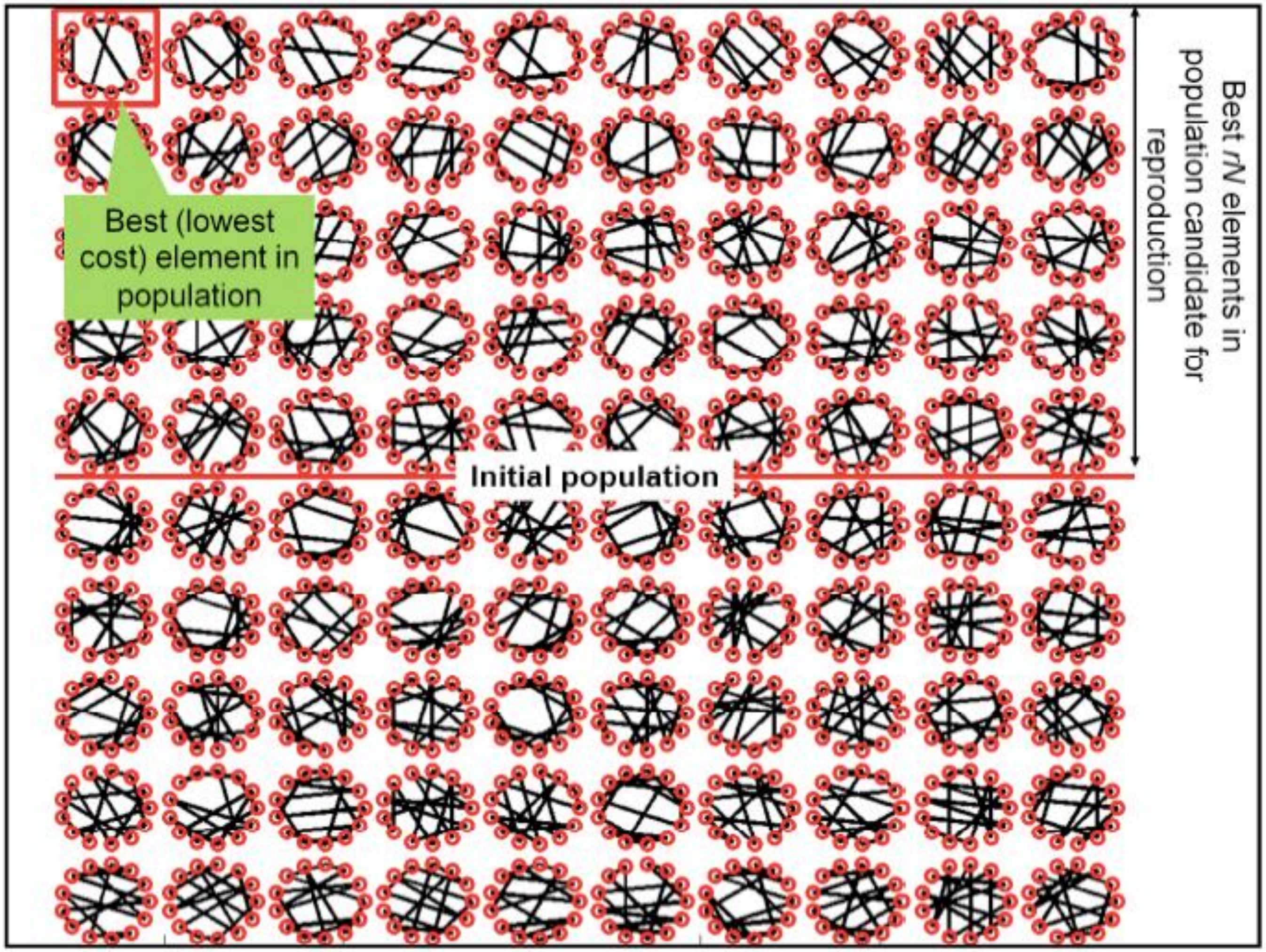


N **= 13**

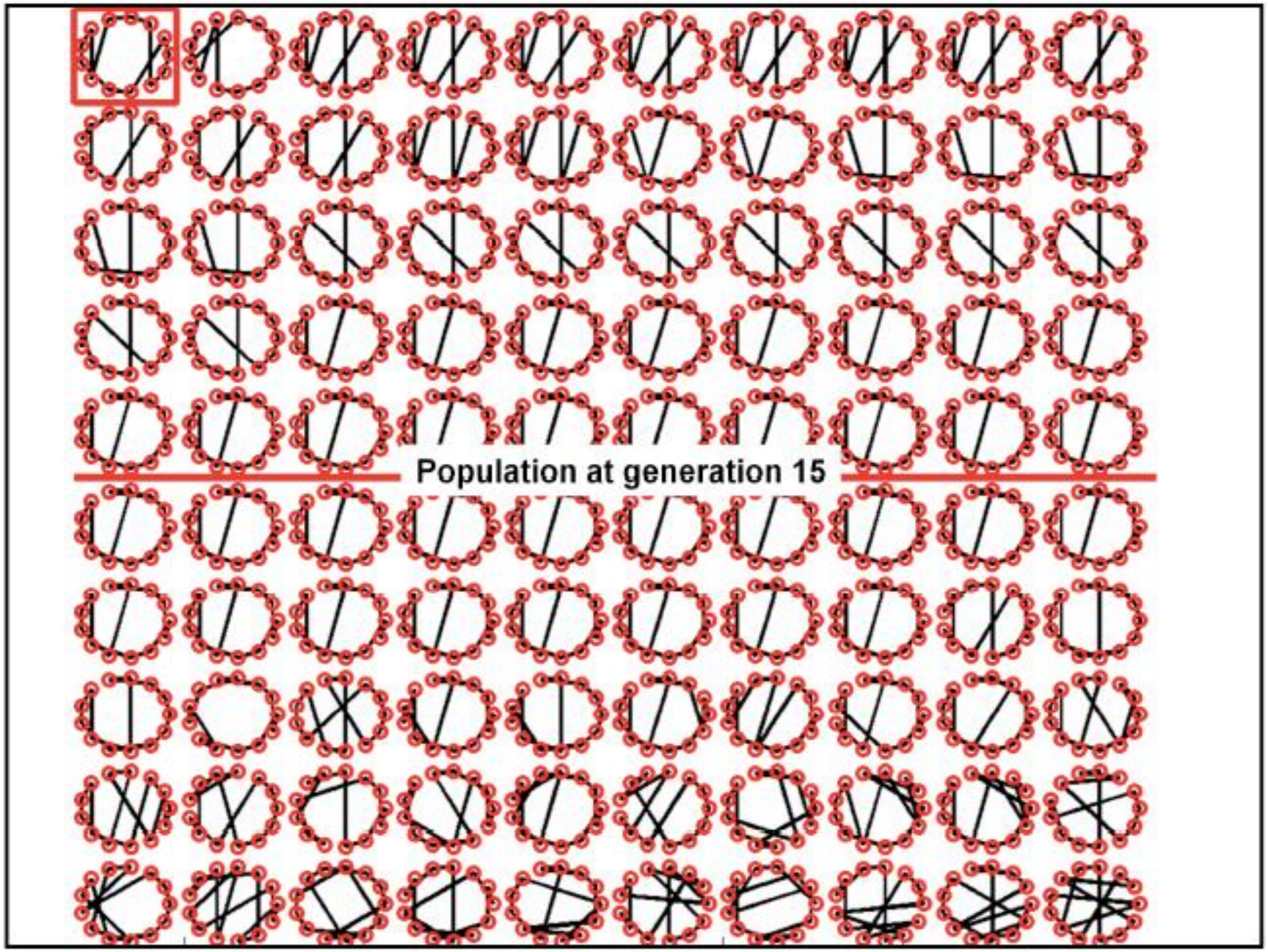
P **= 100 elements in population**

μ = 4% mutation rate
r **= 50% reproduction rate**

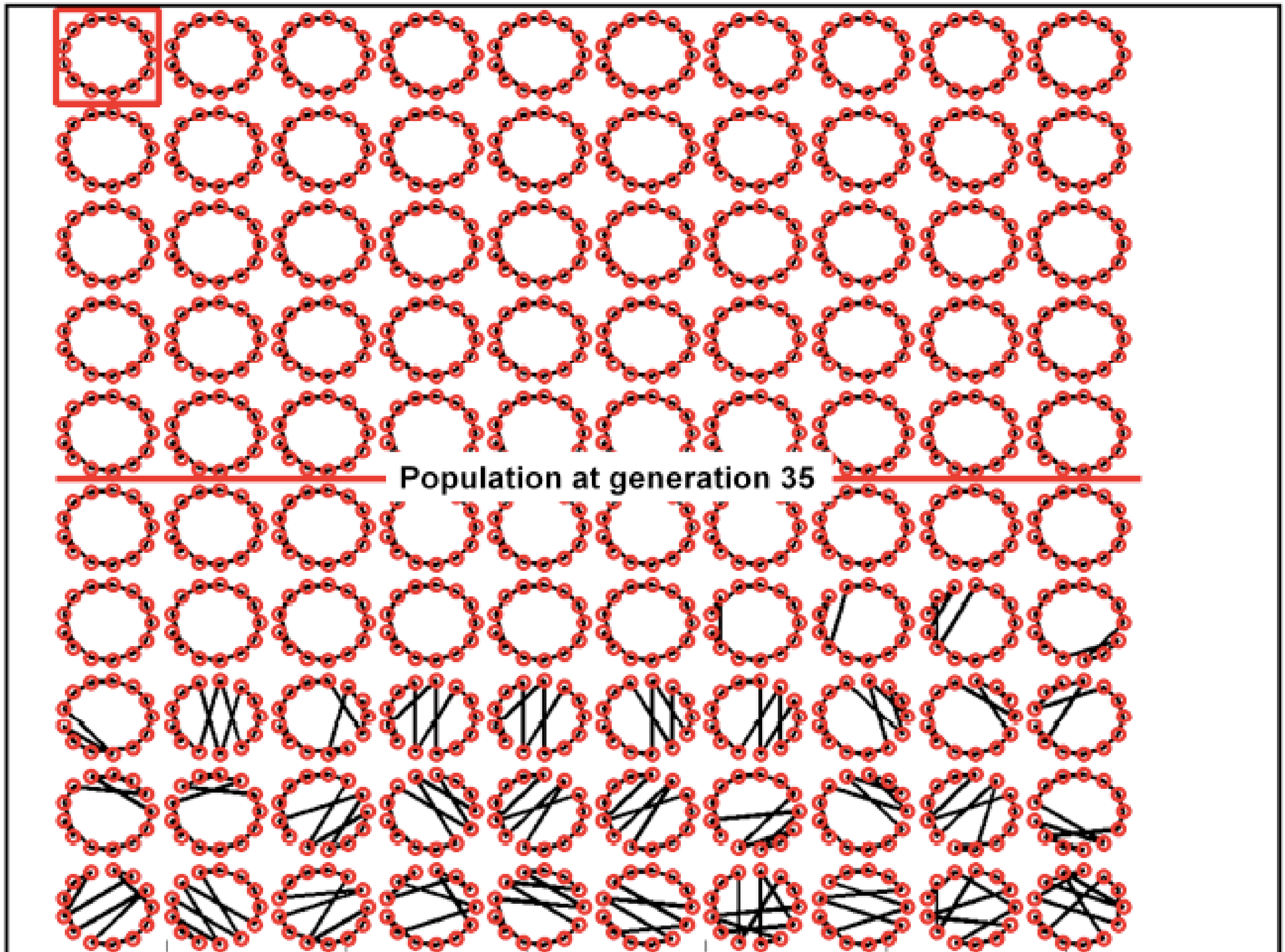
TSP example: Initial generation



TSP example: Generation 15



TSP example: Generation 30



The good and bad of GAs

- Good:
  - Intuitively appealing, due to evolution analogy.
  - If tuned right, can be very effective (good solution with few steps.)
- Bad:
  - Performance depends crucially on the problem encoding. Good encodings are difficult to find!
  - Many parameters to tweak! Bad parameter settings can result in very slow progress, or the algorithm is stuck in local minima.
  - With mutation rate is too low, can get overcrowding (many copies of the identical individuals in the population).