Informed search

使用领域知识

- **Use domain knowledge!**
  - Are we getting close to the goal?
  - Use a heuristic function that estimates how close a state is to the goal
  - A heuristic does NOT have to be perfect!
  - Example of strategies:
    1. Greedy best-first search
    2. A* search
    3. IDA*

启发函数不需要很完美

Greedy search

- Evaluation function $h(n)$ (heuristic)
- $h(n)$ estimates the cost from n to the goal
- Example: $h_{SLD}(n)$ = straight-line distance from n to Sault Ste Marie
- Greedy search expands the node that **appears** to be closest to goal

Greedy search: Pseudo-code

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

→没什么用，因为h(n)是固定的

A* search

- Minimize the total estimated solution cost
- Combines:
  - $g(n)$: cost to reach node n
  - $h(n)$: cost to get from n to the goal
  - $f(n) = g(n)+h(n)$

$f(n)$ is the estimated cost of the cheapest solution through n

有一个中转点

A* search : Pseudo-code

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

与Greedy search相比，仅更改了cost函数.

Admissible heuristics

- An **admissible** heuristic never overestimates the cost to reach the goal, that is it is **optimistic**
- A heuristic h is admissible if

$$\forall node\ n,\ h(n) \leq h^*(n)$$

  where h^* is true cost to reach the goal from n.
- h_{SLD} (used as a heuristic in the map example) is admissible because it is by definition the shortest distance (straight line) between two points.

A* Optimality

If $h(n)$ is admissible, A* using tree search is optimal.

Rationale:

- Suppose G_o is the optimal goal.
  Suppose G_s is some suboptimal goal.
  Suppose n is on the shortest path to Go.
- $f(G_s) = g(G_s)$ since $h(G_s) = 0$
  $f(G_o) = g(G_o)$ since $h(G_o) = 0$
  $g(G_s) > g(G_o)$ since G_s is suboptimal
  Then $f(G_s) > f(G_o)$ . . . (1)
- $h(n) \leq h^*(n)$ since h is admissible
  $g(n)+h(n) \leq g(n)+h^*(n) = g(G_o) = f(G_o)$
  Then, $f(n) \leq f(G_o)$ . . . (2)

From (1) and (2) $f(G_s) > f(n)$, so A* will never select G_s during the search and hence A* is optimal.

We always will go toward Go rather than to go Gs.

G指的是到goal的序列

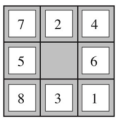
A* : PF Metrics

- **Complete:** Yes.
- **Time:** exponential
- **Space:** keeps every node in memory, the biggest problem
- **Optimal:** Yes!

Search Efficiency of Heuristics

Heuristics for 8-puzzle

- The solution is 26 steps long.
- $h_1(n)$ = number of misplaced tiles
- $h_1(n)$ = 8
- $h_2(n)$ =total Manhattan distance (sum of the horizontal and vertical distances).
- Tiles 1 to 8 in the start state gives: h_2 = 3+1+2+2+2+3+3+2 = 18 which does not overestimate the true solution.



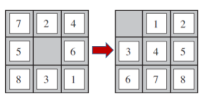
Start State

Goal State

- $h_{mis}(s)$ = #misplaced titles ∈ [0,8]: **Admissible**.
- $h_{1stp}(s)$ = #(1-step move) to reach the goal configuration: **Admissible**.

➢ $h_{1stp}(s) \geq h_{mis}(s) \Rightarrow h_{1stp}(s)$ is **'better'** than $h_{mis}(s)$.

What does 'better' mean?



Dominance

- For **admissible** h_1 and h_2, if $h_1(s) \geq h_2(s)$ for $\forall s$
  $\Rightarrow h_1$ **dominates** h_2 and is **more efficient** for search.
- **Theorem**: For any admissible heuristics h_1 and h_2, define
  $$h(s) = \max\{h_1(s), h_2(s)\}$$
  $h(s)$ is admissible and dominates both h_1 and h_2.

取大的能防止过多搜索，提高效率

- **'Better' heuristic = dominance = better search efficiency.**
- **Question**: Which one to choose from a collection of admissible heuristics $h_1, \cdots, h_m$ & none dominates any other?
- **Answer**: $h(s) = \max\{h_1(s), \cdots, h_m(s)\}$ dominates all the others.

Quantify Search Efficiency

- **Effective Branching Factor b^***: For a solution from A*, calculate b^* satisfying: $N = b^* + (b^*)^2 + \cdots + (b^*)^d$
  - N: #nodes of the solution,
  - d: depth of the solution tree.
  - E.g., A* finds a solution at depth 5 using 52 nodes ⇒ $b^* = 1.92$.
- Good heuristics have b^* close to 1 ⇒ large problems solved at reasonable computational cost.
- **b^* quantifies search efficiency of heuristics.**

希望b^*越接近1越好

Empirical: Factor b^*

- **Aim**: Compare h_1 and h_2 regarding the search efficiency.
- **Setting**: Generate 1200 random problems with $d = \{2, \cdots, 24\}$ and solve them with IDS and A* with h_1 & h_2.
- **Note**: IDS – a baseline.

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

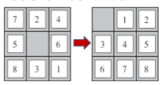
- h_2 is **'better'** than h_1 regarding **search efficiency**.
- This **goodness** is reflected by **b^* being closer to 1**.
- A* with h_2 performs much better than IDS.

Generate Admissible Heuristics

① from Relaxed Problems

For 8-puzzle problem:

- **Real Rule**: A tile can only move to the **adjacent empty** square.
- **Relaxed rules**: h_{mis} and h_{1stp} are admissible
  - R1: A tile can move **anywhere** ⇒ $h_{mis}(s)$ = #(misplaced titles).
  - R2: A tile can move one step in **any direction** regardless of an occupied neighbour ⇒ $h_{1stp}(s)$ = #(1-step move) to reach goal.
- **Optimal solutions to problems with R1, R2 are easier to find.**



Relaxed Problem

- **Relaxed problem**: a problem with **relaxed rules** on the action.
- E.g. 8-puzzle problems with R1 and R2.

- **Theorem**: The cost of an optimal solution to **a relaxed problem** is an **admissible heuristic** for the original problem.
- No wonder h_{mis} and h_{1stp} are admissible.

② from sub-problems

- **Subproblem**
  - **Task**: get tiles 1, 2, 3 and 4 into their correct positions.
  - **Relaxation**: move them disregarding the others.
- **Theory**: cost*(subproblem)<cost*(original).
  - cost*(subproblem): the cost of the optimal solution of this subproblem.

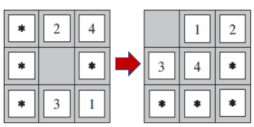


Fig.1. A subproblem of 8-puzzle.

Subproblem and Addmissible Heuristics

- **Admissible $h^*_{sub}(s)$**: estimate the cost from s to the subproblem goal.
  - E.g. $h^{(1,2,3,4)}_{sub}$ is the cost to solve the 1-2-3-4 subproblem.

- **Theorem**: $h_{sub}(s)$ dominates $h_{1stp}(s)$,
  - $h_{sub}(s) = max\{h^{(1,2,3,4)}_{sub}(s), h^{(2,3,4,5)}_{sub}(s), \cdots\}$.

Disjoint Subproblems

- **Question**: Will the **addition of heuristics** from subproblem (1-2-3-4) and (5-6-7-8) give an **admissible heuristic**, considering the two subproblems are not overlapped?
- **Answer**: No, since they always **share some moves**.

- **Question**: What if **not count** those shared moves?
- **Answer**: $h^{(1,2,3,4)}_{sub}(s) + h^{(5,6,7,8)}_{sub}(s) \le c^*(s)$ ⇒ admissible.
  - Disjoint pattern database

③ from Experience

For 8-puzzle problem:

- Solve many 8-puzzles to obtain **many examples**.
- Each **example** consists of a state from the solution path and the actual cost of the solution from that point.
- These **examples** are our '**experience**' for this problem.

- **Question**: How to learn $h(s)$ from these **experience**?

Learn Heuristics from Experience

- **Question**: What are the **good experience features**?
- **Answer**: **Relevant** to predicting the states' cost to Goal, e.g.
  - $x_1(s)$: #(displaced tiles).
  - $x_2(s)$: #(pairs of adjacent tiles) that are not adjacent in Goal state.
- **Question**: How to learn h from those **relevant experience features**?
- **Answer**: (e.g.) Construct model as

$$h(s) = w_1 x_1(s) + w_2 x_2(s),$$

where w_1, w_2 are model parameters to learn from training data by a learning method such as neural networks and decision trees.

权值可以是经验生成的

Search Methods

- **Uniformed search**: Use **no** domain knowledge.
  - BFS, DFS, DLS, IDS, UCS
- **Informed search**: Use a heuristic function that **estimates** how close a state is **to the goal**.
  - Greedy search, A*, IDA*.

We can organize the algorithms into pairs where the first proceeds by layers, and the other proceeds by subtrees.

(1) Iterate on Node Depth:

- BFS searches layers of increasing node depth.
- IDS searches subtrees of increasing node depth.

(2) Iterate on Path Cost + Heuristic Function:

- A* searches layers of increasing path cost + heuristic function.
- IDA* searches subtrees of increasing path cost + heuristic function.

Which cost function?

- UCS searches layers of increasing path cost.
- Greedy best first search searches layers of increasing heuristic function.
- A* search searches layers of increasing path cost + heuristic function.

Local Search

- Search algorithms seen so far are designed to **explore search spaces systematically.**
- Problems: observable, deterministic, known environments
- where the solution is a sequence of actions.
- Real-World problems are more complex.
- When a goal is found, the path to that goal constitutes a solution to the problem. But, depending on the applications, **the path may or may not matter**.
- If the path does not matter/systematic search is not possible, then consider another class of algorithms.
- In such cases, we can use iterative improvement algorithms, **Local search**.
- Also useful in pure **optimization problems** where the goal is to find the best state according to an **optimization function**.
- **Examples:**
  - Integrated circuit design, telecommunications network optimization, etc.
  - 8-queen: what matters is the final configuration of the puzzle, not the intermediary steps to reach it.

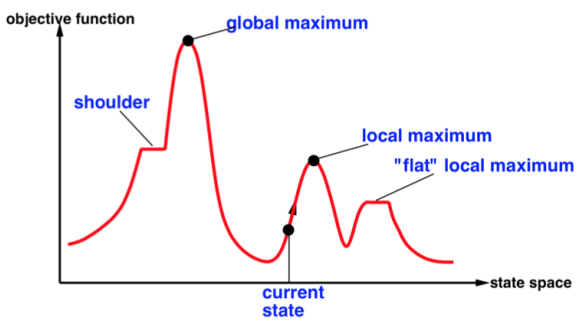
目标是找到最优状态，根据最优函数

- **Idea**: keep a single "current" state, and try to improve it.
- Move only to neighbors of that node.
- **Advantages**:
  1. No need to maintain a search tree.
  2. Use very little memory.
  3. Can often find good enough solutions in continuous or large state spaces.

只关注当前状态并优化它，只选择该状态附近的状态

- **Local Search Algorithms:**
  - Hill climbing (steepest ascent/descent).
  - Simulated Annealing: inspired by statistical physics.
  - Local beam search.
  - Genetic algorithms: inspired by evolutionary biology.

爬山法
模拟退火
局部搜索
遗传算法



State space landscape

Hill climbing

- Also called **greedy local search**.
- Looks only to immediate good neighbors and not beyond.
- Search moves uphill: moves in the direction of increasing elevation/value to find the top of the mountain.
- Terminates when it reaches a **peak**.
- Can terminate with a local maximum, global maximum or can get stuck and no progress is possible.
- A **node is a state and a value**.

仅看最优邻居
顺着使评估值增大的方向不断前进直到山顶.
当到顶峰时停止.
容易停在局部最优

Pseudo-code

```
function HILL-CLIMBING(initialState)
    returns State that is a local maximum

    initialize current with initialState

    loop do
        neighbor = a highest-valued successor of current

        if neighbor.value ≤ current.value:
            return current.state

        current = neighbor
```

Variants

Other variants of hill climbing include

- **Sideways moves** escape from a plateau where best successor has same value as the current state.
- **Random-restart** hill climbing overcomes local maxima: keep trying! (either find a goal or get several possible solution and pick the max).
- **Stochastic** hill climbing chooses at random among the uphill moves.

- **Hill climbing** effective in general but depends on shape of the landscape. Successful in many real-problems after a reasonable number of restarts.
- **Local beam search** maintains *k* states instead of one state. Select the *k* best successor, and useful information is passed among the states.
- **Stochastic beam search** choose *k* successors are random. Helps alleviate the problem of the states agglomerating around the same part of the state space.

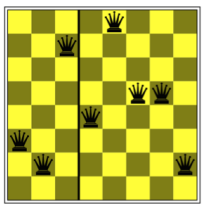
Stochastic：做下一步的选择是随机的
beam：批量选择

Genetic algorithms

- **Genetic algorithm (GA)** is a variant of stochastic beam search.
- Successor states are generated by combining two parents rather by modifying a single state.
- The process is inspired by **natural selection**.
- Starts with *k* **randomly generated states**, called **population**. Each state is an **individual**.
- An individual is usually represented by a **string** of 0's and 1's, or digits, a finite set.
- The objective function is called **fitness function**: better states have high values of fitness function.

自然选择
一开始有k个状态，即个体，个体合起来称为种群。
个体通常由0、1字符串来表示。
适应性函数：评估个体的优劣

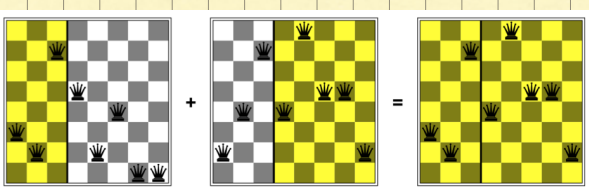
- In the 8-queen problem, an individual can be represented by a **string** digits 1 to 8, that represents the position of the 8 queens in the 8 columns.



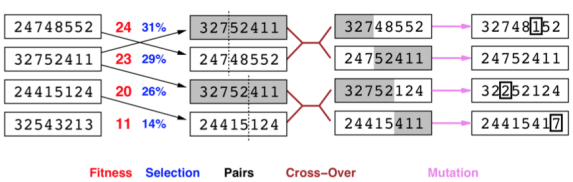
41

- The objective function is called **fitness function**: better states have high values of fitness function.
- Possible fitness function is the **number of non-attacking pairs of queens.**
- Fitness function of the solution: 28.
- Pairs of individuals are selected at random for **reproduction** w.r.t. some probabilities.
- A **crossover** point is chosen randomly in the string.
- **Offspring** are created by crossing the parents at the crossover point.
- Each element in the string is also subject to some **mutation** with a small probability.

交配产生后代时，双方均从自己的字符串中截取一部分生成新个体，交叉点是随机选择的。
后代会有一定概率产生突变



Generate successors from pairs of states.



Pseudo-code

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Simulated Annealing

- Hill Climbing → Simulated Annealing

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
    neighbor ← a highest-valued successor state of current
    if VALUE(neighbor) ≤ VALUE(current) then return current
    current ← neighbor
```

爬山法是贪婪局部搜索，容易陷入局部最优/高原

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current ← problem.INITIAL
  for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE(current) – VALUE(next)
    if ΔE < 0 then current ← next      //for maximization
    else current ← next only with probability e^{−ΔE/T}
```

模拟退火是允许下山的爬山法.

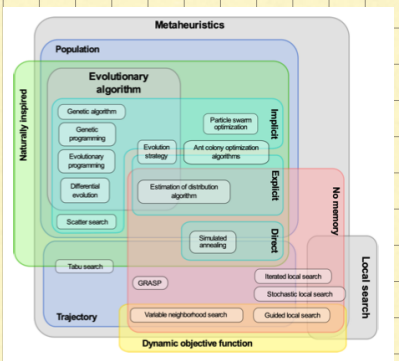
Further Studies on Metaheuristics

Comparison

Heuristic	Metaheuristic
• Problem-specific	• Problem-independent
• Can be used by a metaheuristic	• Can use different heuristics or a combination of heuristics

元启发算法是与问题无关的

Metaheuristics



Reproduction (crossover)

Mutation

"One general law, leading to the advancement of all organic beings, namely, multiply, vary, let the strongest live and weakest die."

- Charles Darwin, *The Origin of Species*

Selection

Evolutionary Computation

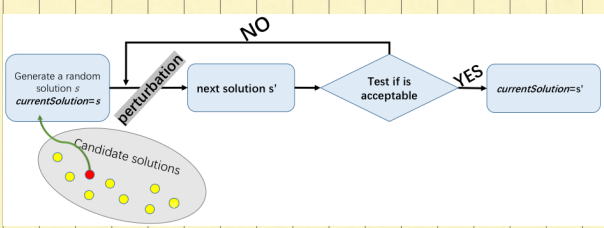
- It is the study of computational systems which use ideas and get inspirations from natural evolution.
- One of the principles borrowed is ***survival of the fittest***.
- Evolutionary computation (EC) techniques can be used in optimisation, learning, and design.
- EC techniques do **not** require rich domain knowledge to use. However, domain knowledge can be incorporated into EC techniques.

适者生存

演化计算(EC

EC算法无需大量的领域知识，但可以使用在EC算法中.

Generate-and-Test (G&T)

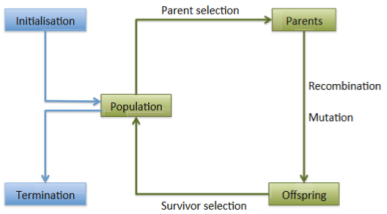


Steps

1. Generate the initial solution at random and denote it as the **current solution**.
2. Generate the **next solution** from the current one by **perturbation**.
3. Test whether the newly generated solution (**next solution**) is acceptable;
   1. Accepted it as the current solution **if yes**;
   2. Keep the current solution unchanged **otherwise**.
4. Go to Step 2 if the current solution is not satisfactory, stop otherwise.

EA: Population-based G&T

- **Generate**: Mutate and/or recombine individuals in a population.
- **Test**: Select the next generation from the parents and offspring.



A Simple Evolutionary Algorithm (EA)

1 Generate the initial population $P(0)$ at random
2 $i \leftarrow 0$ *// Generation counter*
3 **WHILE** halting criteria are not satisfied
4 **Evaluate** the fitness of each individual in $P(i)$
5 **Select** parents from $P(i)$ based on their fitness in $P(i)$
6 **Generate** offspring from the parents using **crossover** and **mutation** to form $P(i+1)$
7 $i \leftarrow i+1$

So how does this simple EA work?

Illustration Example

Let's use the simple **EA with population size 4** to maximise the function

$$f(x) = x^2$$

with x in the *integer* interval $[0, 31]$, i.e., $x = 0, 1, \ldots, 30, 31$.

- Population size = 4 ⇔ 4 individuals/chromosomes
- So, what is an **individual** or **chromosome**?

Example
Encoding and decoding

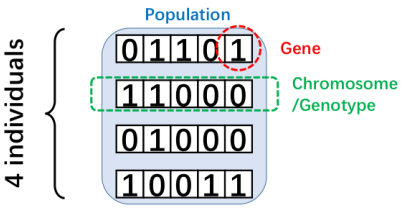
- **Representation**: The first step of EA applications is ***encoding*** (i.e., the representation of chromosomes).
  - We adopt binary representation for integers.
  - 5 bits are used to represent integers up to 31.
  - Examples:

31 **Encoding** → | 1 | 1 | 1 | 1 | 1 | **Decoding** → 31 **Evaluation** → 961

16 **Encoding** → | 1 | 0 | 0 | 0 | 0 | **Decoding** → 16 **Evaluation** → 196

EA : Step 1

1. **Initialisation**: Generate initial population at random, e.g., 01101, 11000, 01000, 10011. These are ***chromosomes*** or ***genotypes***.



EA: Step 2

2. **Evaluation**: Calculate fitness value for each individual.

   a) **Decode** the individual into an integer (called ***phenotypes***):

   01101 -> 13; 11000 -> 24, 01000 -> 8, 10011 -> 19;

   b) **Evaluate** the fitness according to $f(x) = x^2$:

   $$f(13) = 169, f(24) = 576, f(8) = 64, f(19) = 361.$$

EA: Step 3-a

3. **Crossover**:
   a) Select two individuals for crossover based on their fitness. If **roulette-wheel selection** is used, then

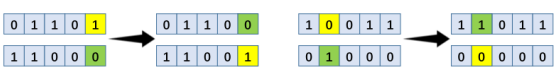
$$P_i = \frac{f_i}{\sum_j f_j}$$

Two offspring are often produced and added to an intermediate population. Repeat this step until the intermediate population is filled.
In our example:

$P_1(13) = \frac{169}{1170} = 0.14,\ P_2(24) = \frac{576}{1170} = 0.49,\ P_3(8) = \frac{64}{1170} = 0.06,\ P_4(19) = \frac{361}{1170} = 0.31$

EA: Step 3-b

b) Examples of **crossover**

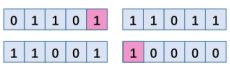


Now the intermediate population is 01100, 11001, 11011, 00000.

EA: Steps 4-5

4. Apply **mutation** to individuals in the intermediate population with a ***small*** probability. A simple mutation is bit-flipping. For example, we may have the following new population $P(1)$ after random mutation:

Example:



5. Go to step 2 if not stop.

Different Evolutionary Algorithms

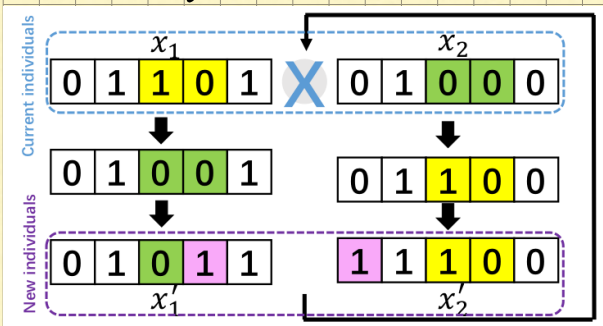
- There are several well-known EAs with different
  - historical backgrounds,
  - representations,
  - variation operators,
  - and selection schemes.

In fact, EAs refer to **a whole family of algorithms**, not a single algorithm.

EA families

- Genetic Algorithms (GAs)
- Evolutionary Programming (EP)
- Evolution Strategies (ES)
- Genetic Programming (GP)
- ...

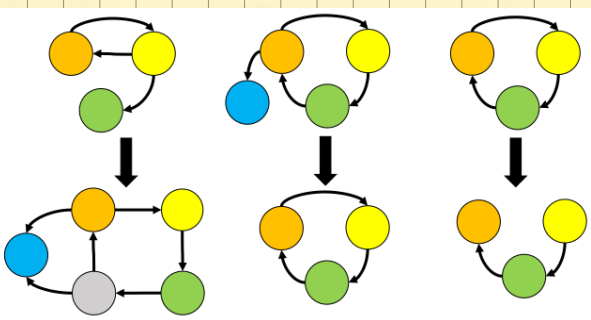
Genetic Algorithms (GAs)



- First formulated by Holland for adaptive search and by his students for optimisation from mid 1960s to mid 1970s.
- Binary strings have been used extensively as individuals (*chromosomes*).
- Simulate **Darwinian evolution**.
- Search operators are only applied to the *genotypic* representation (chromosome) of individuals.
- Emphasise the role of **recombination** (*crossover*). Mutation is only used as a background operator.
- Often use **roulette-wheel** selection.

Representation	Bit-strings
Recombination	1-Point crossover
Mutation	Bit flip
Parent selection	Fitness proportional - implemented by Roulette Wheel
Survival selection	Generational

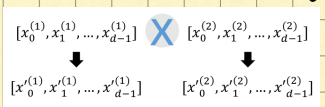
Evolutionary Programming (EP)



- First proposed by Fogel *et al.* in mid 1960s for simulating intelligence.
- **Finite state machines** (FSMs) were used to represent individuals, although real-valued vectors have always been used in numerical optimisation.
- It is closer to **Lamarckian evolution**.
- Search operators (mutations only) are applied to the ***phenotypic*** representation of individuals.
- It does ***not*** use any recombination.
- Usually use **tournament** selection.

有限状态机

Evolution Strategies (ES)

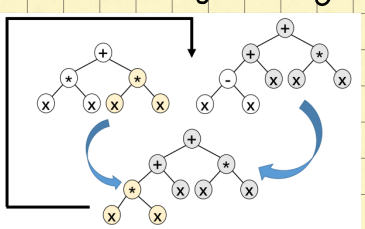


- First proposed by Rechenberg and Schwefel in mid 1960s for numerical optimisation.
- **Real-valued vectors** are used to represent individuals.
- They are closer to **Larmackian evolution**.
- They do have recombination.
- They use ***self-adaptive mutations***.

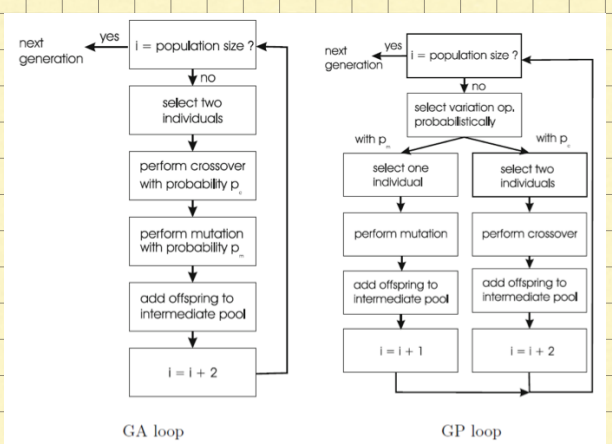
Representation	Real-valued vectors
Recombination	Discrete or intermediary
Mutation	Gaussian perturbation
Parent selection	Uniform random
Survivor selection	Deterministic elitist replacement by (μ, λ) or $(\mu + \lambda)$
Speciality	Self-adaptation of mutation step sizes

- μ: parents size
- λ: offspring size

Genetic Programming (GP)



- First used by de Garis to indicate the evolution of artificial neural networks, but used by Koza to indicate the evolution of computer programs.
- **Trees** (especially Lisp expression trees) are often used to represent individuals.
- Both ***crossover*** and ***mutation*** are used.



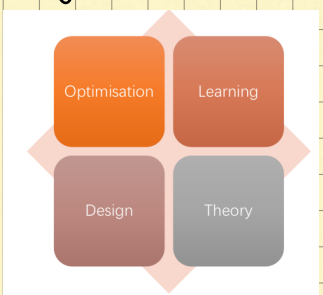
Preferred Term: Evolutionary Algorithms

- EAs face the same fundamental issues as those classical AI faces, i.e., representation, and search.
- Although GAs, EP, ES, and GP are different, they are all different variants of **population-based** **generate-and-test** algorithms. They share more similarities than differences!
- A better and more general term to use is evolutionary algorithms (EAs).

Variants in Operators

- **Crossover/Recombination**: one-point crossover, two-point crossover, uniform crossover, intermediate crossover, etc.
- **Mutation**: bit-flipping, Gaussian mutation, Cauchy mutation, etc.
- **Selection**: roulette wheel selection (fitness proportional selection), tournament selection, rank-based selection (linear and nonlinear), etc.
- **Replacement Strategy**: generational, steady-state (continuous), etc.
- **Specialised Operators**: multi-parent recombination, inversion, order-based crossover, etc.

Major Areas in EC



Evolutionary Optimization

- Numerical (global) optimisation.
- Combinatorial optimisation (of NP-hard problems).
- Mixed optimisation.
- Constrained optimisation.
- Multi-objective optimisation.
- Optimisation in a dynamic environment (with a dynamic fitness function).

Evolutionary Learning

Evolutionary learning can be used in supervised, unsupervised and reinforcement learning.

- Learning classifier systems (Rule-based systems).
- Evolutionary artificial neural networks.
- Evolutionary fuzzy logic systems.
- Co-evolutionary learning.

Evolutionary Design

EC techniques are particularly good at exploring unconventional designs which are very difficult to obtain by hand.

- Evolutionary design of artificial neural networks.
- Evolutionary design of electronic circuits.
- Evolvable hardware.
- Evolutionary design of (building) architectures.

Summary

- Evolutionary algorithms can be regarded as population-based generate-and-test algorithms.
- Evolutionary computation techniques can be used in optimisation, learning and design.
- Evolutionary computation techniques are flexible and robust.
- Evolutionary computation techniques are definitely useful tools in your toolbox, but there are problems for which other techniques might be more suitable.