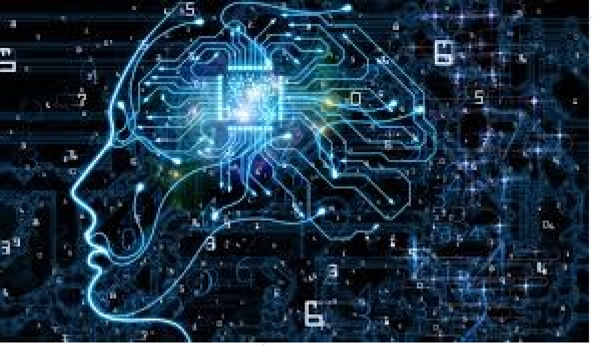


Chapter-3

Local Search and Constraints Satisfaction problem

Local Search

Local search methods work on complete state formulations. They keep only a small number of nodes in memory.

Local search is useful for solving optimization problems:

- o Often it is easy to find a solution
- o But hard to find the best solution

Algorithm goal:

find optimal configuration (e.g., TSP),

- ❑ Hill climbing
- ❑ Gradient descent
- ❑ Simulated annealing

• For some problems the state description contains all of the information relevant for a solution. Path to the solution is unimportant.

• Examples:

- o map coloring
- o 8-queen

Hill Climbing Algorithm

Very simple idea: Start from some state *s*,

- ❑ Move to a neighbor ***t*** with better score. Repeat.

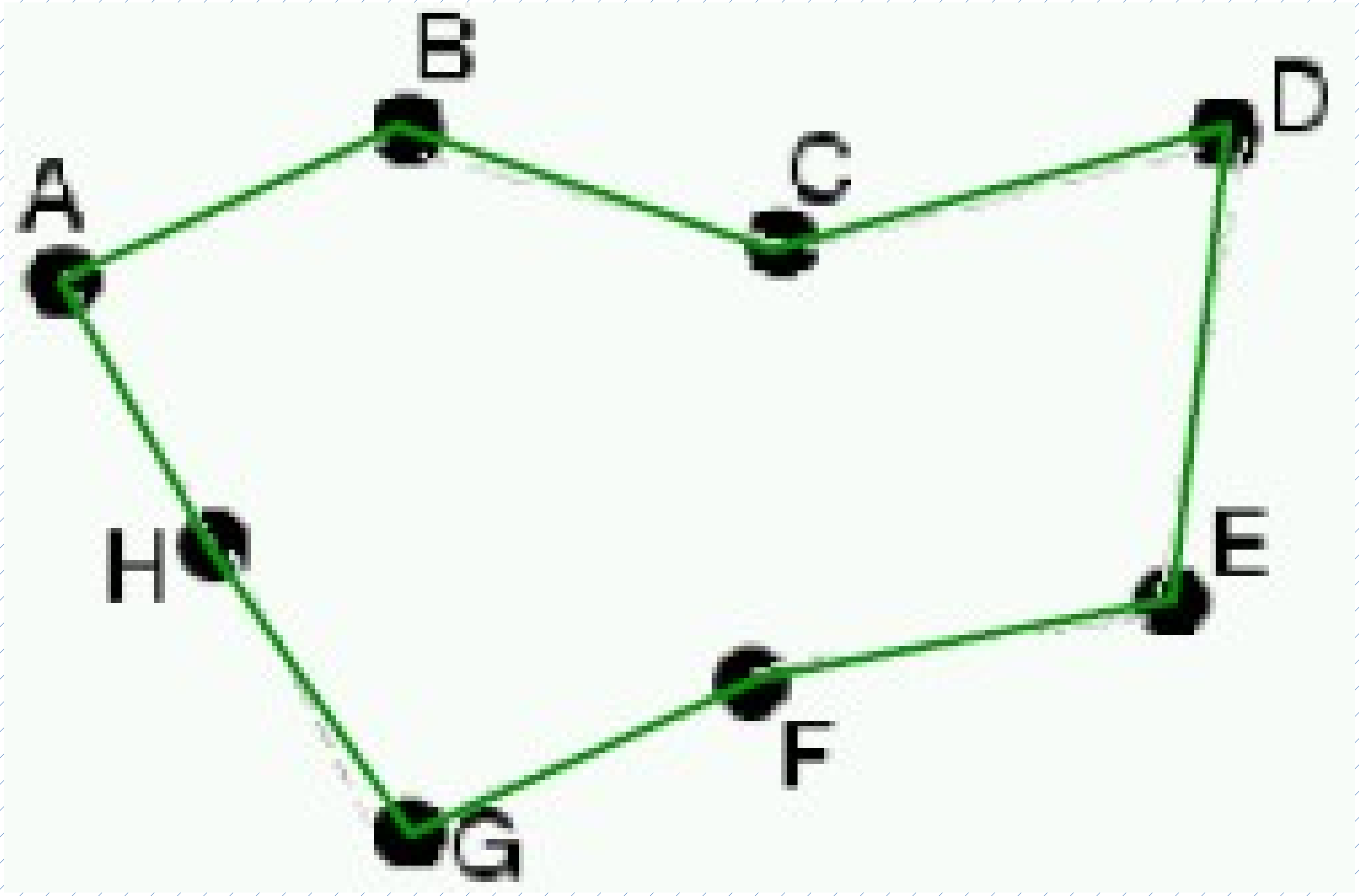
Question: what's a neighbor?

You have to define that!

- ❑ The neighborhood of a state is the set of neighbors
- ❑ Also called 'move set'
- ❑ Similar to successor function

Neighbors: TSP

- state: A-B-C-D-E-F-G-H-A
- *f* = length of tour



Hill Climbing Algorithm

Question: What’s a neighbor?

- ❑ Problems tend to have structures. A small change produces a neighboring state.
- ❑ The neighborhood must be small enough for efficiency
- ❑ Designing the neighborhood is critical. This is the real ingenuity – not the decision to use hill climbing.

Question: Pick which neighbor?

Question: What if no neighbor is better than the current state?

Hill Climbing Algorithm

1. Pick initial state s
2. Pick t in neighbors(s) with the largest $f(t)$
3. IF $f(t) \le f(s)$ THEN stop, return s
4. $s = t$. GOTO 2.

Not the most sophisticated algorithm in the world.

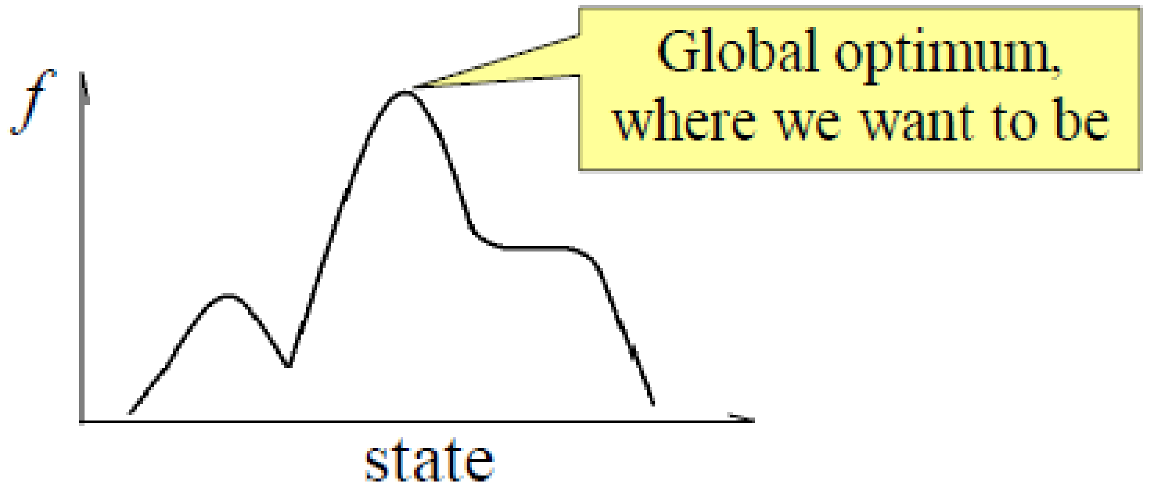
Very greedy.

Easily stuck.

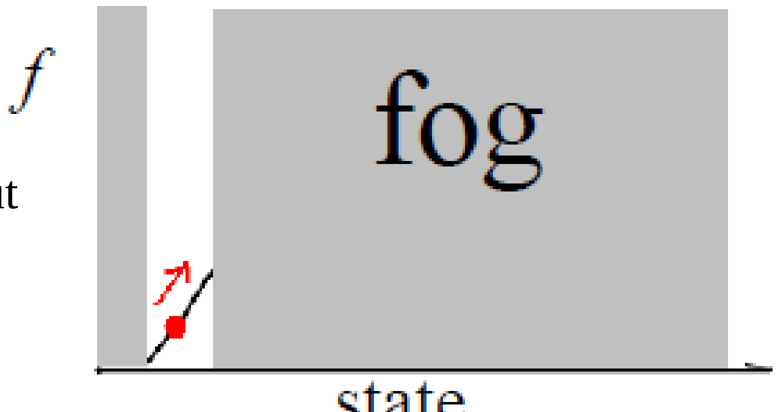
your enemy:
local optima

Local optima in hill climbing

- Useful conceptual picture: *f* surface = 'hills' in state space



- But we can't see the landscape all at once. Only see the neighborhood. Climb in fog.



The rest of the lecture is about **Escaping local optima**

Variation of Hill Climbing

Question: How do we make hill climbing less greedy?

Stochastic hill climbing

- Randomly select among better neighbors
- The better, the more likely
- Pros / cons compared with basic hill climbing?

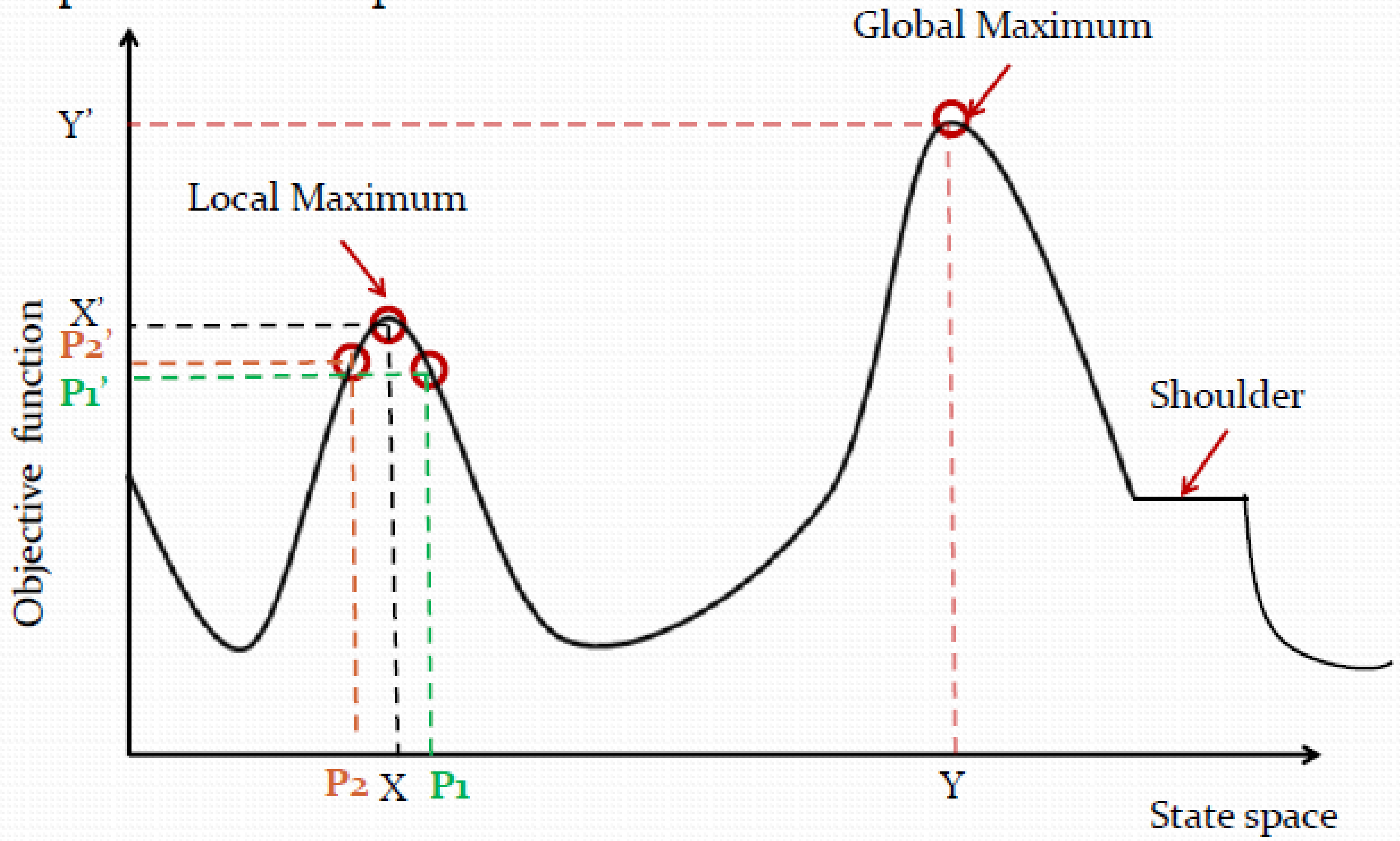
- **Question**: What if the neighborhood is too large to enumerate? (e.g. N-queen if we need to pick both the column and the move within it)

First-choice hill climbing

- Randomly generate neighbors, one at a time
- If better, take the move
- Pros / cons compared with basic hill climbing?

Hill Climbing Algorithm

- Local Search Algorithm
- State space landscape



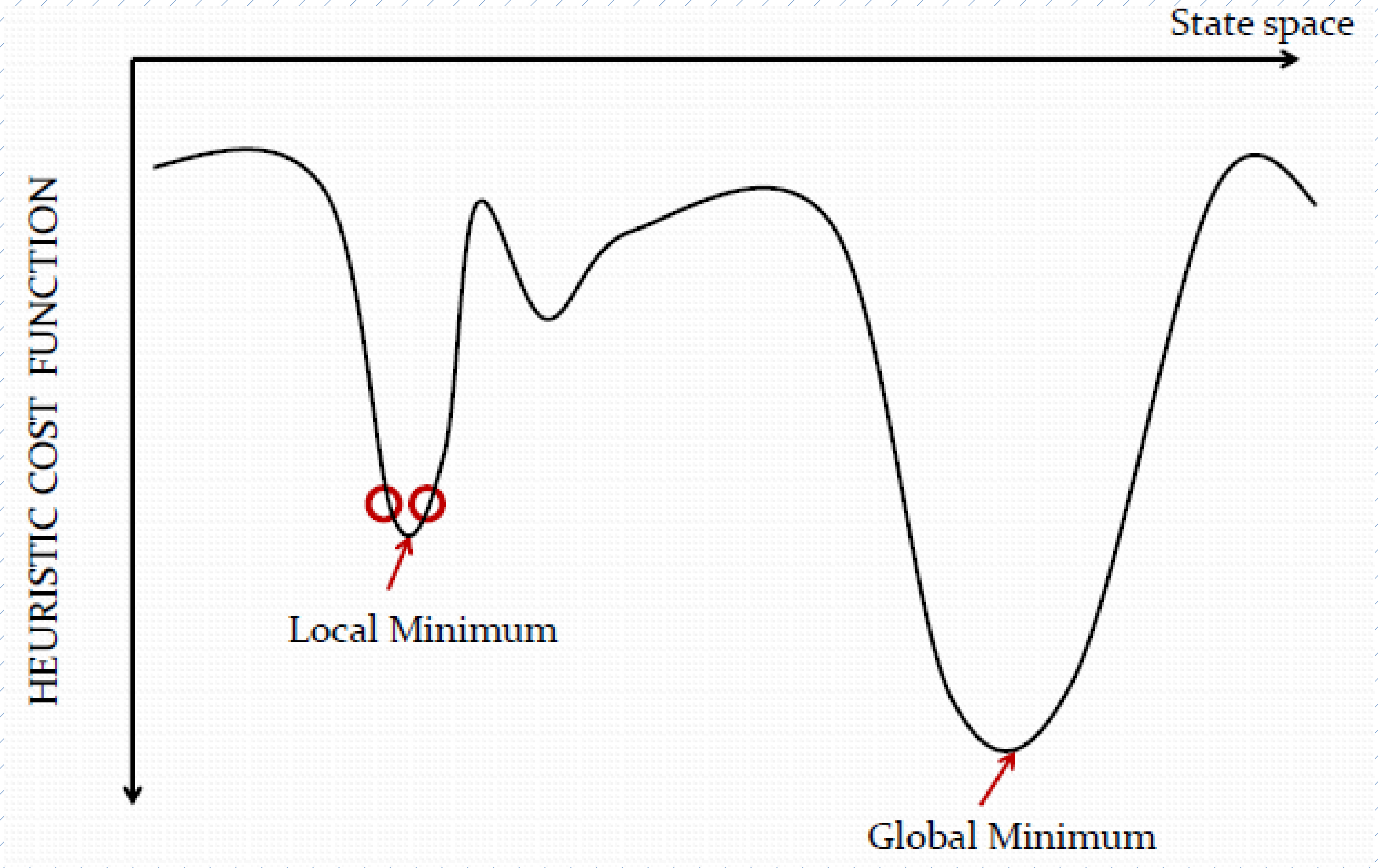
Hill Climbing Algorithm

Maximization function is called Objective Function

Minimization function is called Heuristic Cost function

Heuristic cost= distance, time, money spent

Objective function= profit, success



Hill Climbing Algorithm

1. Evaluate the initial state. If it is the goal state then return and quit. Otherwise continue with initial state as current state.
2. Loop until a solution is found or until there are no new operators left to be applied to the current state:

a. Select operator that has not been applied to the current state and apply it to produce the new state.

b. Evaluate the new state

  i. If it is the goal state, then return and quit

  ii. If it is not a goal state but it is better than the current state then make it the current state.

  iii. If it is not better than the current state then continue in the loop.

Hill Climbing Algorithm Problems

Local maxima

Once the top of a hill is reached the algorithm will halt since every possible step leads down.

Plateaux

If the landscape is flat, meaning many states have the same goodness, algorithm degenerates to a random walk.

Ridges

If the landscape contains ridges, local improvements may follow a zigzag path up the ridge, slowing down the search.

Advantages of Hill Climbing

It can be used in continuous as well as discrete domains.

Disadvantages of Hill Climbing

1. Not efficient method –not suitable to problems where the value of
heuristic function drops off suddenly when solution may be in sight.
2.Local search method- gets caught up in local maxima/minima.

Solution to Local Maxima problem:

1. Simulated annealing
. Backtracking to some earlier node and try different direction.

SIMULATED ANNEALING

Anneal

To subject (glass or metal) to a process of heating and slow cooling in order to toughen and reduce brittleness.

1.Pick initial state *s*

2.Randomly pick *t* in neighbors(*s*)

3.IF *f*(*t*) better THEN accept $s \leftarrow t$.

4.ELSE /* *t* is worse than *s* */

5. accept $s \leftarrow t$ with a small probability

6.GOTO 2 until bored.

SIMULATED ANNEALING

- How to choose the small probability?
- idea 1: $p = 0.1$
- idea 2: p decreases with time
- idea 3: p decreases with time, also as the 'badness' $|f(s)-f(t)|$ increases
- If $f(t)$ better than $f(s)$, always accept t
- Otherwise, accept t with probability

$$\exp\left(-\frac{|f(s)-f(t)|}{Temp}\right)$$

Boltzmann distribution

SIMULATED ANNEALING

- If *f*(*t*) better than *f*(*s*), always accept *t*
- Otherwise, accept *t* with probability

$$\exp\left(-\frac{|f(s)-f(t)|}{Temp}\right)$$

Boltzmann distribution

Temp is a temperature parameter that 'cools' (anneals) over time, e.g. Temp $\leftarrow$ Temp*0.9 which gives Temp=(T0)#iteration

- High temperature: almost always accept any *t*
- Low temperature: first-choice hill climbing

- If the 'badness' (formally known as energy difference) |*f*(*s*)-*f*(*t*)| is large the

SA Algorithm

```
assuming we want to maximize f()
current = Initial-State(problem)
for t = 1 to œ do
T = Schedule(t) ; // T is the current temperature, which is monotonically decreasing with t
if T=0 then return current ; //halt when temperature = 0
next = Select-Random-Successor-State(current) deltaE = f(next) - f(current) ; // If positive, next is better than current. Otherwise, next is worse than current.
if deltaE > 0 then current = next ; // always move to a better state
else current = next with probability p = exp(deltaE / T) ; // as $T \to 0$,
 $p \to 0$; as deltaE  -  , p  0
end
```

SA Algorithm Design Issues

Cooling scheme important

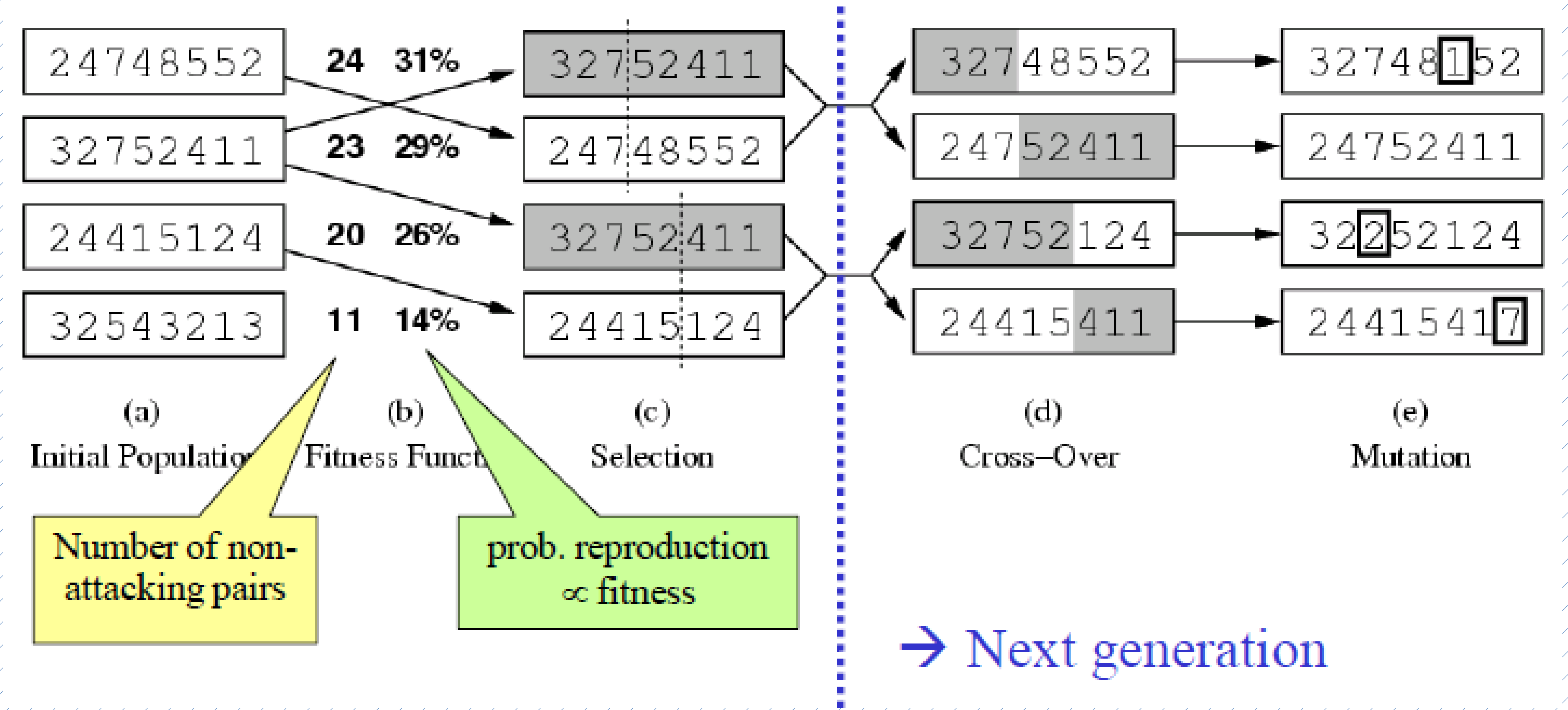
- Neighborhood design is the real ingenuity, not the decision to use simulated annealing.
- Not much to say theoretically

With infinitely slow cooling rate, finds global optimum with probability 1.

- Proposed by **Metropolis** in 1953 based on the analogy that alloys manage to find a near global minimum energy state, when annealed slowly.
- Easy to implement.
- Try hill-climbing with random restarts first!

Genetic algorithm

Genetic algorithm: a special way to generate neighbors, using the analogy of **cross-over, situutation, and natural selection**.

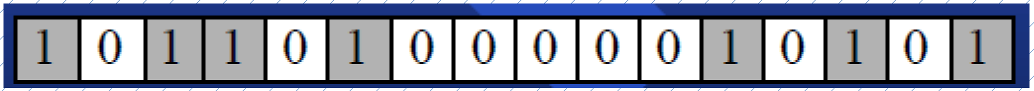


Genetic Algorithms

In the early 1970s, John Holland introduced the concept of genetic algorithms.

His aim was to make computers do what nature does. Holland was concerned with algorithms that manipulate strings of binary digits.

Each artificial “chromosomes” consists of a number of “genes”, and each gene is represented by 0 or 1:



Genetic Algorithms

- ❑ Nature has an ability to adapt and learn without being told what to do. In other words, nature finds good chromosomes blindly. GAs do the same. Two mechanisms link a GA to the problem it is solving: **encoding** and **evaluation**.
- ❑ The GA uses a measure of fitness of individual chromosomes to carry out reproduction. As reproduction takes place, the crossover operator exchanges parts of two single chromosomes, and the mutation operator changes the gene value in some randomly chosen location of the chromosome.

Basic Genetic Algorithms

Step 1: Represent the problem variable domain as
a chromosome of a fixed length, choose the size
of a chromosome population *N*, the crossover
probability *pc* and the mutation probability *pm*.

Step 2: Define a fitness function to measure the
performance, or fitness, of an individual
chromosome in the problem domain. The

Basic Genetic Algorithms

Step 3: Randomly generate an initial population of

chromosomes of size *N*:

$x1, x2, \ldots, xN$

Step 4: Calculate the fitness of each individual

chromosome:

$f(x1), f(x2), \ldots, f(xN)$

Step 5: Select a pair of chromosomes for mating

from the current population. Parent

chromosomes are selected with a

Basic Genetic Algorithms

Step 6: Create a pair of offspring chromosomes by

applying the genetic operators – **crossover** and

mutation.

Step 7: Place the created offspring chromosomes

in the new population.

Step 8: Repeat *Step 5* until the size of the new

chromosome population becomes equal to the

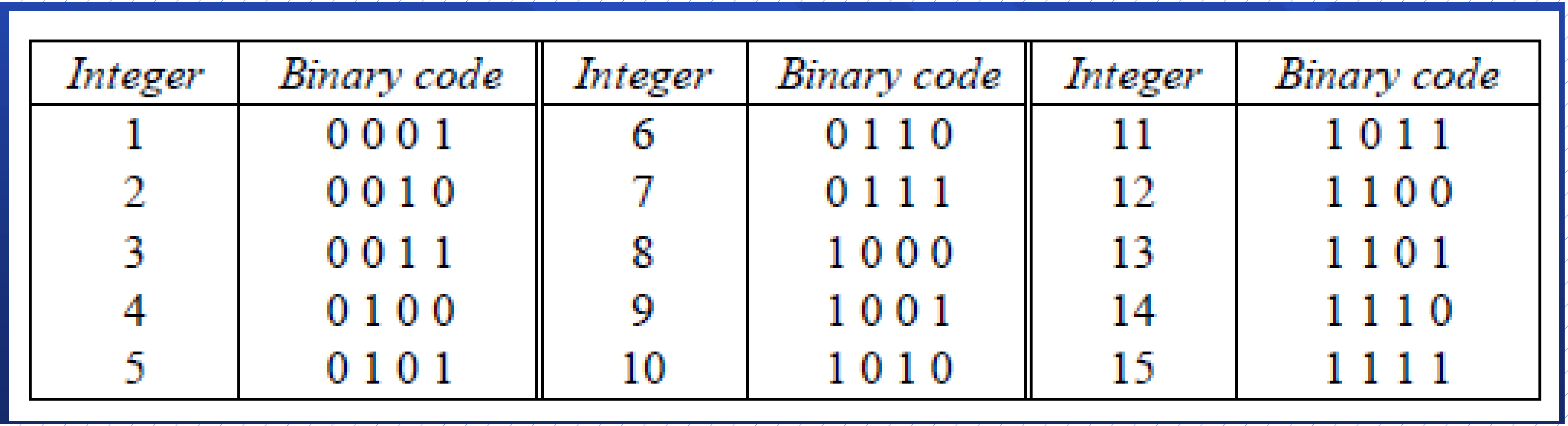
Genetic Algorithms

GA represents an iterative process. Each iteration is
called a **generation**. A typical number of generations
for a simple GA can range from 50 to over 500. The
entire set of generations is called a **run**.

Because GAs use a stochastic search method, the
fitness of a population may remain stable for a
number of generations before a superior

case study

A simple example will help us to understand how a GA works. Let us find the maximum value of the function (15x − x2) where parameter x varies between 0 and 15. For simplicity, we may assume that x takes only integer values. Thus, chromosomes can be built with only four genes:



Integer	Binary code	Integer	Binary code	Integer	Binary code
1	0 0 0 1	6	0 1 1 0	11	1 0 1 1
2	0 0 1 0	7	0 1 1 1	12	1 1 0 0
3	0 0 1 1	8	1 0 0 0	13	1 1 0 1
4	0 1 0 0	9	1 0 0 1	14	1 1 1 0
5	0 1 0 1	10	1 0 1 0	15	1 1 1 1

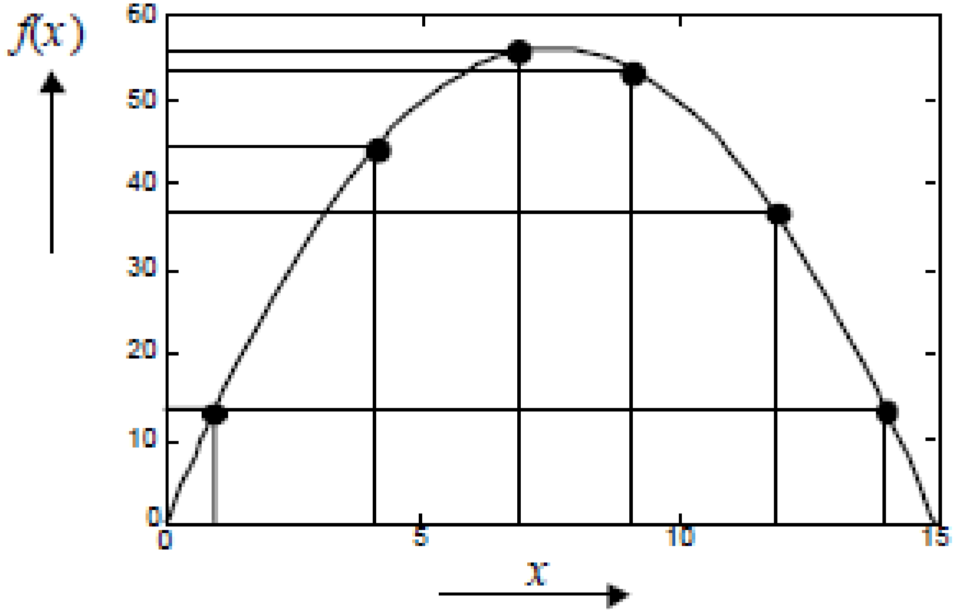
Genetic algorithms: case study

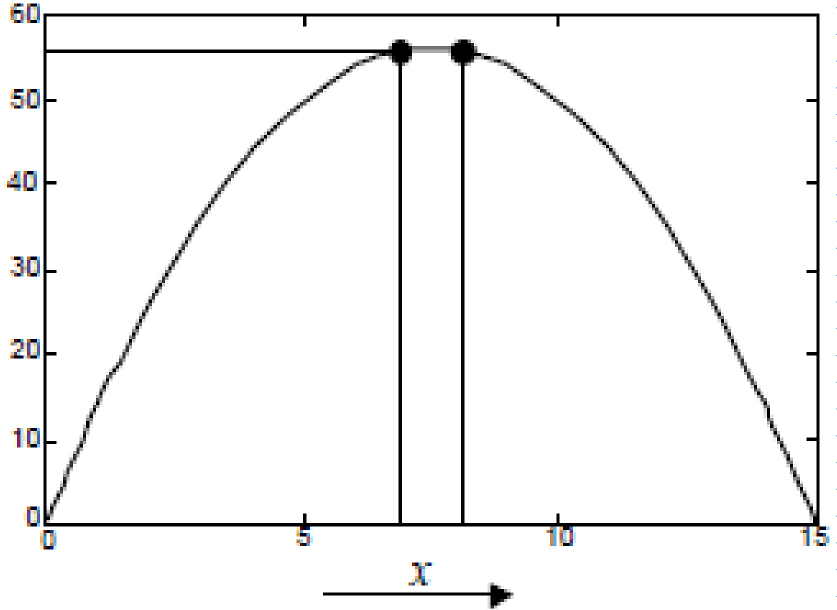
Suppose that the size of the chromosome population *N* is 6, the crossover probability *pc* equals 0.7, and the mutation probability *pm* equals 0.001. The fitness function in our example is defined by

$$\mathbf{f(x) = 15\,x - x2}$$

The fitness function and chromosome locations

Chromosome label	*Chromosome string*	*Decoded integer*	*Chromosome fitness*	*Fitness ratio, %*
X1	1 1 0 0	12	36	16.5
X2	0 1 0 0	4	44	20.2
X3	0 0 0 1	1	14	6.4
X4	1 1 1 0	14	14	6.4
X5	0 1 1 1	7	56	25.7
X6	1 0 0 1	9	54	24.8



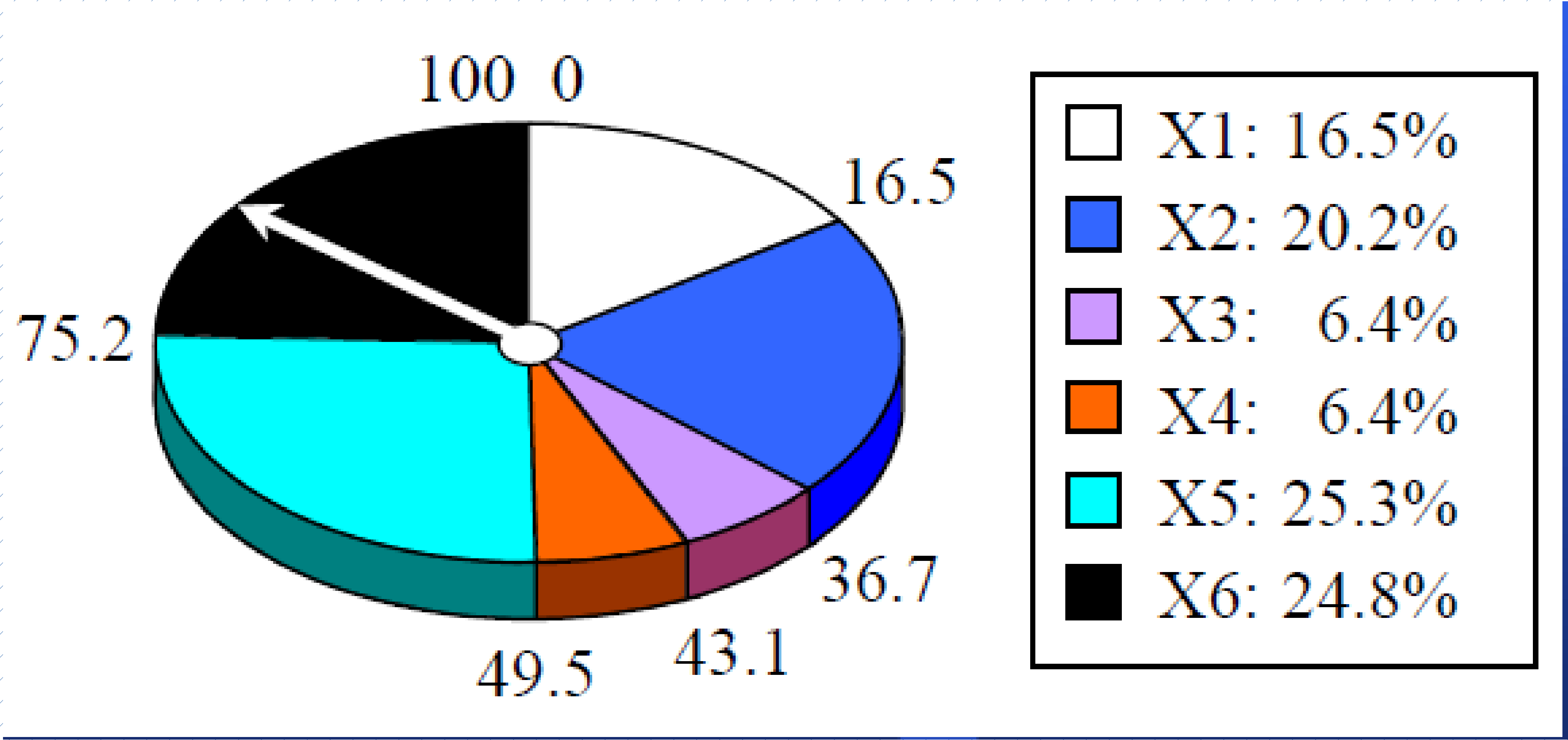


In natural selection, only the fittest species can survive, breed, and thereby pass their genes on to the next generation. GAs use a similar approach, but unlike nature, the size of the chromosome population remains unchanged from one generation to the next.

The last column in Table shows the ratio of the individual chromosome’s fitness to the population’s total fitness. This ratio determines the chromosome’s chance of being selected for mating. The chromosome’s average fitness improves from one generation to the next.

Roulette wheel selection

The most commonly used chromosome selection techniques is the **roulette wheel selection**.



Crossover operator

In our example, we have an initial population of 6

chromosomes. Thus, to establish the same

population in the next generation, the roulette

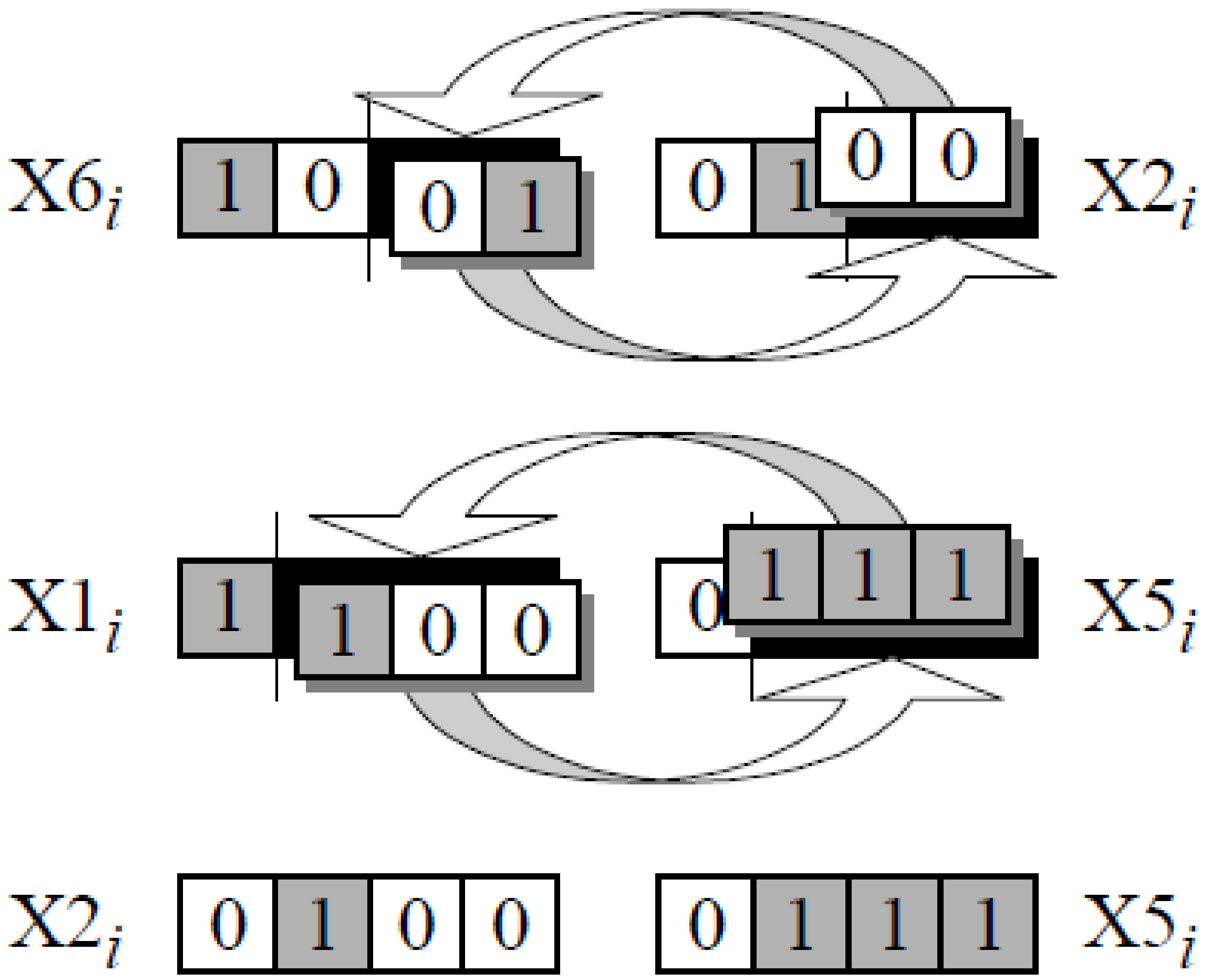
wheel would be spun six times.

Once a pair of parent chromosomes is selected,

the **crossover** operator is applied.

First, the crossover operator randomly chooses a

Crossover operator



Mutation operator

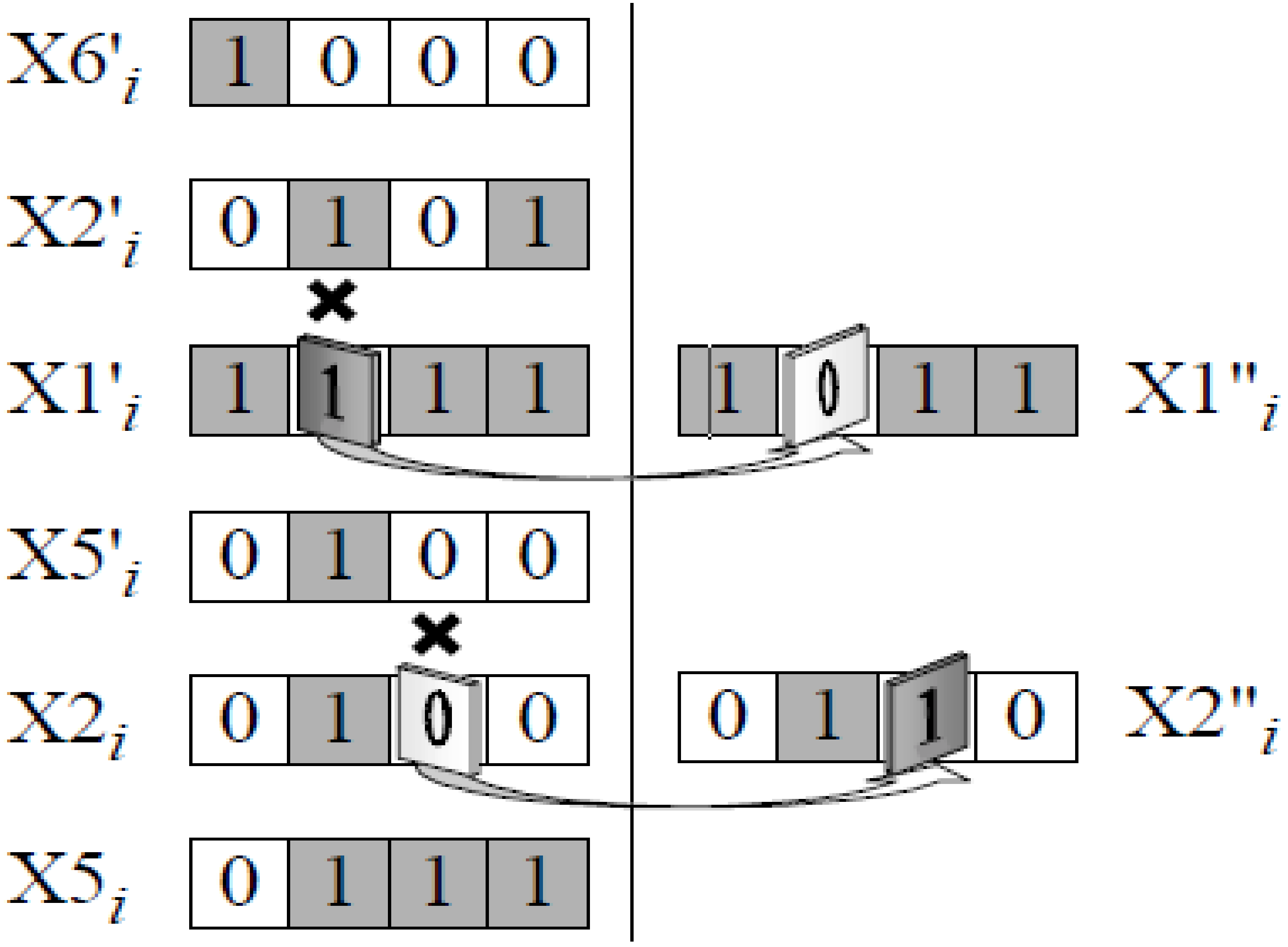
Mutation represents a change in the gene.

Mutation is a background operator. Its role is to
provide a guarantee that the search algorithm is
not trapped on a local optimum.

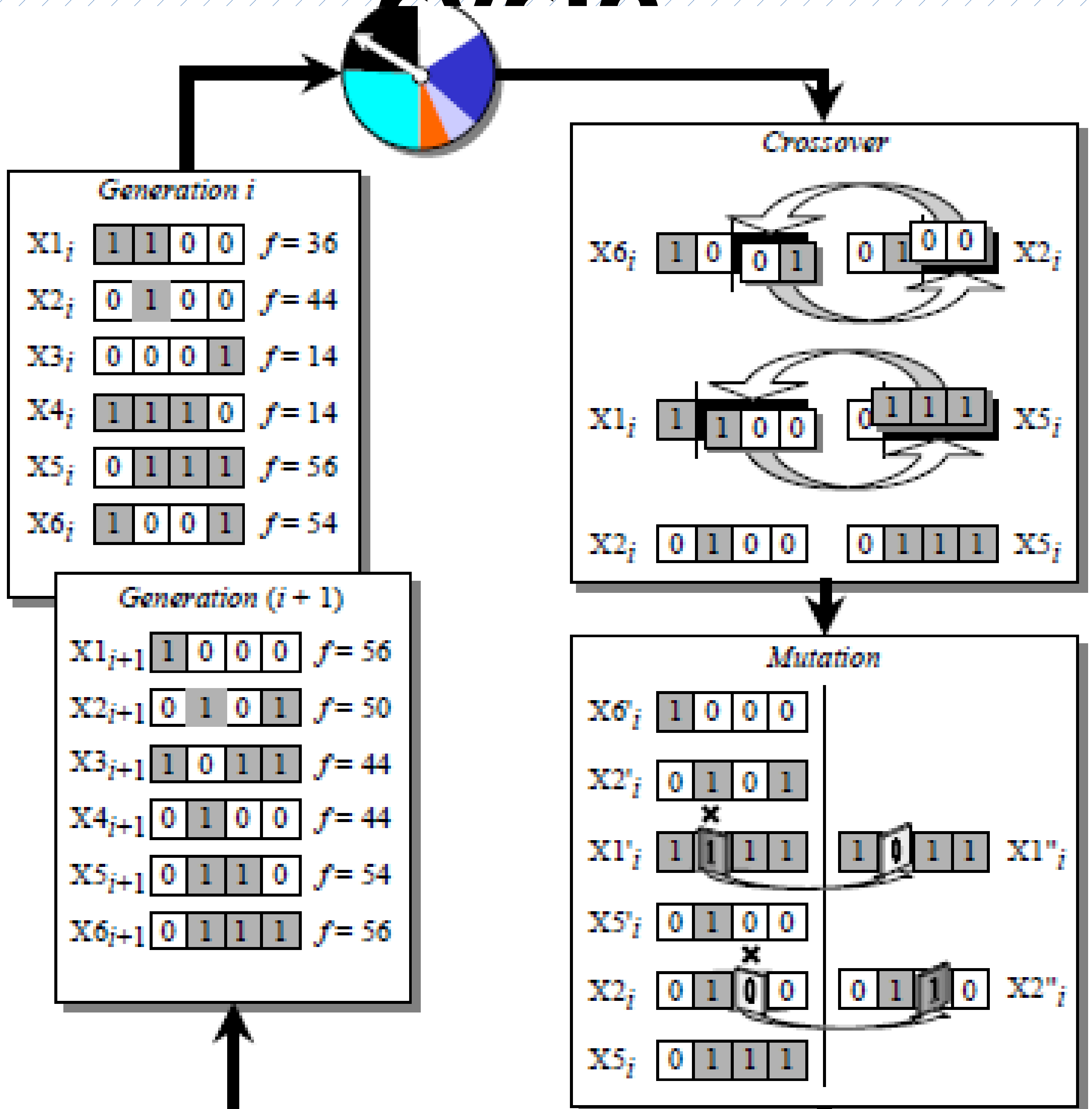
The mutation operator flips a randomly selected
gene in a chromosome.

The mutation probability is quite small in nature

Mutation operator



The genetic algorithm cycle



Steps in the GA development

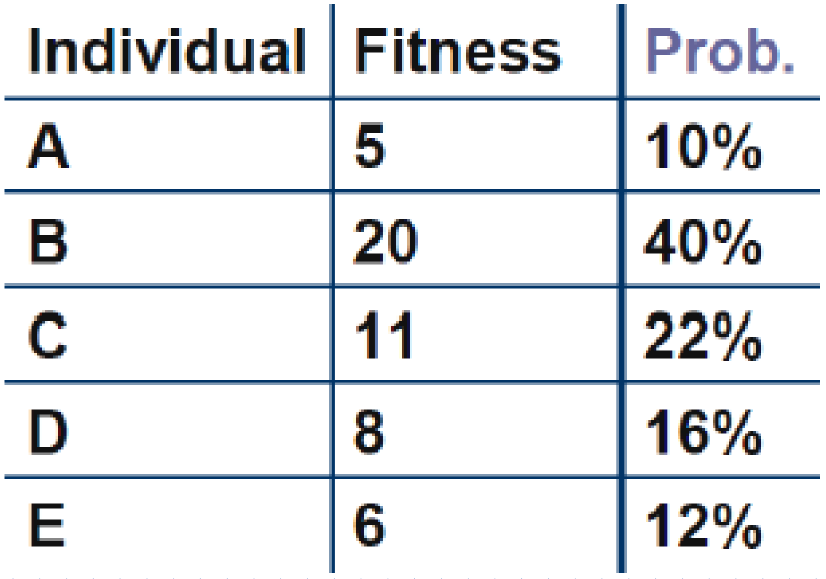
1. Specify the problem, define constraints and optimum criteria;
2. Represent the problem domain as a chromosome;
3. Define a fitness function to evaluate the chromosome performance;
4. Construct the genetic operators;
5. Run the **GA** and tune its parameters.

Genetic algorithm

1. Let *s1*, ..., *sN* be the current population
2. Let $p_i = f(s_i)/\sum_j f(s_j)$ be the reproduction probability
3. FOR $k = 1$; $k<N$; $k+=2$
   - ❑parent1 = randomly pick according to *p*
   - ❑parent2 = randomly pick another
   - ❑randomly select a crossover point, swap strings of parents 1, 2 to generate children $t[k]$, $t[k+1]$
4. FOR $k = 1$; $k<=N$; $k++$
   - •Randomly mutate each position in $t[k]$ with a small probability (mutation rate)
5. The new generation replaces the old: $\{s\} \leftarrow \{t\}$. Repeat.

Proportional selection

- $p_i = f(s_i) / \Sigma_j f(s_j)$
- $\Sigma_j f(s_j) = 5+20+11+8+6=50$
- $p_1=5/50=10\%$



Individual	Fitness	Prob.
A	5	10%
B	20	40%
C	11	22%
D	8	16%
E	6	12%

Variations of genetic algorithm

- ❑ Parents may survive into the next generation
- ❑ Use ranking instead of *f*(*s*) in computing the reproduction probabilities.
- ❑ Cross over random bits instead of chunks.
- ❑ Optimize over sentences from a programming language.

Constraint Satisfaction Problems

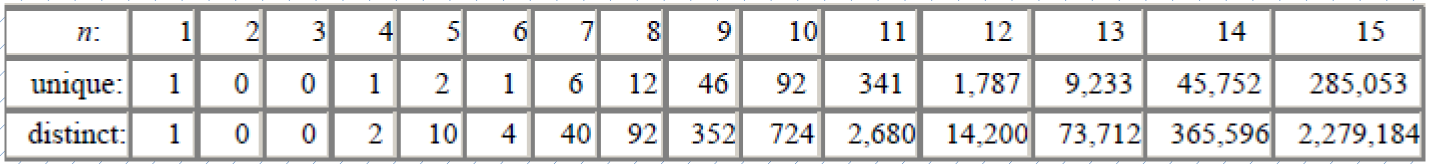
Constraint satisfaction problems or **CSP**s are mathematical problems where one must find states or objects that satisfy a number of constraints or criteria. A constraint is a restriction of the feasible solutions in an optimization problem

Many problems can be stated as constraints satisfaction problems. *Here are some examples*

Example 1: The n-Queen problem is the problem of putting n chess queens on an *n*×*n* chessboard such that none of them is able to capture any other using the standard chess queen's moves. The color of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The problem was originally proposed in 1848 by the chess player Max Bazzel, and over the years, many mathematicians, including Gauss have worked on this puzzle. In 1874, S. Gunther proposed a method of finding solutions by using determinants, and J.W.L. Glaisher refined this approach

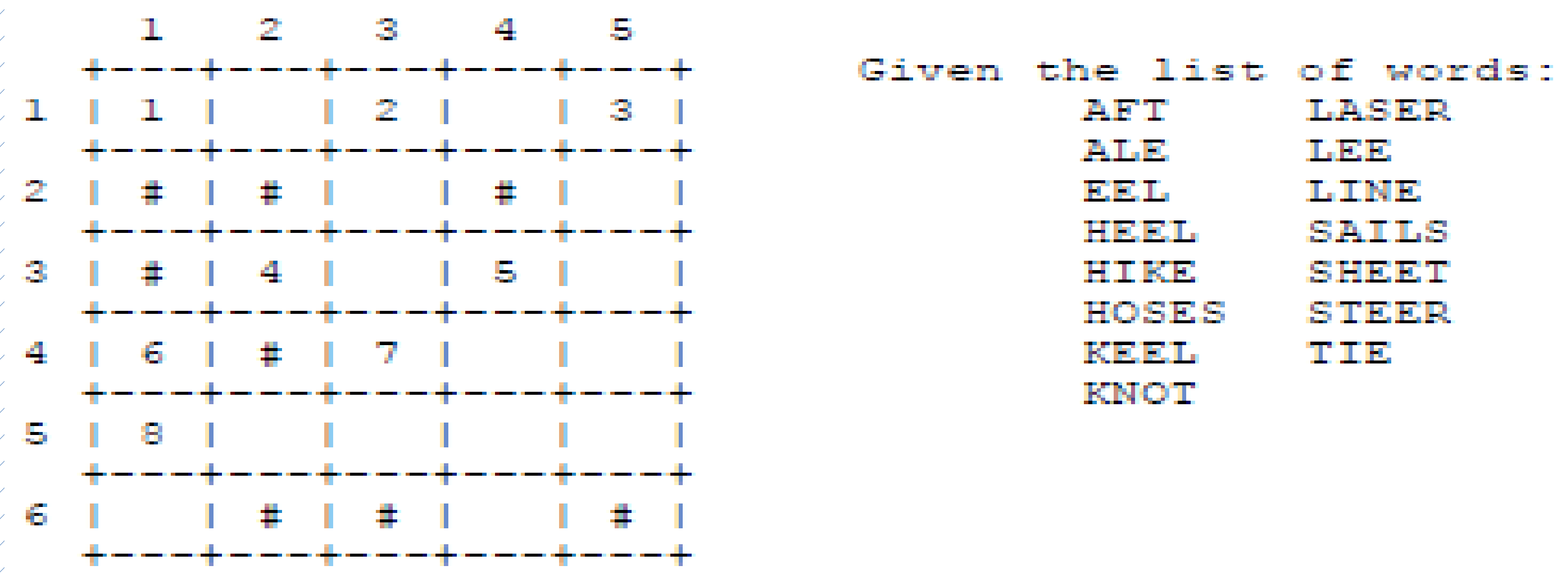
The eight queens puzzle has 92 **distinct** solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 **unique** solutions. The following table gives the number of solutions for *n* queens, both unique and distinct.



n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
unique:	1	0	0	1	2	1	6	12	46	92	341	1,787	9,233	45,752	285,053
distinct:	1	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	2,279,184

Note that the 6 queens puzzle has, interestingly, fewer solutions than the 5 queens puzzle!

Example 2: A crossword puzzle: We are to complete the puzzle



```
      1    2    3    4    5
   +----+----+----+----+----+
1  | 1  |    | 2  |    | 3  |
   +----+----+----+----+----+
2  | #  | #  |    | #  |    |
   +----+----+----+----+----+
3  | #  | 4  |    | 5  |    |
   +----+----+----+----+----+
4  | 6  | #  | 7  |    |    |
   +----+----+----+----+----+
5  | 8  |    |    |    |    |
   +----+----+----+----+----+
6  |    | #  | #  |    | #  |
   +----+----+----+----+----+
```

```
Given the list of words:
       AFT       LASER
       ALE       LEE
       EEL       LINE
       HEEL      SAILS
       HIKE      SHEET
       HOSES     STEER
       KEEL      TIE
       KNOT
```

We’ve seen CSP before!

Constraint satisfaction problem (CSP) is a special
class of search problem
Each problem has a set of variables (e.g. A,B,C,D,E)
• Each variable take a value from a domain (e.g. {T,F})
• Each problem has a set of constraints (e.g A □ □B □ C=T)
• Objective: find a complete assignment of variables that
satisfies all the constraints.
• What are v/v/d/c of 8-queen? Map coloring?

CSP definition

- CSP is a triplet {*V*, *D*, *C*}
- • *V* = {*V*1, *V*2, …, *V*n} a finite set of variables
- • Each variable may be assigned a value from domain *D*i
- • Each member of C is a pair
-   First member: a subset of variables
-   Second member: a set of valid values
- • Example:
- *V* = {*V*1, *V*2, …, *V*7}
- *D* = {R, G, B}
- *C* = { (*V*1, *V*2):{(R,G), (R,B), (G,B), (G,R), (B,G), (B,R)},
- (*V*1, *V*3):{(R,G), (R,B), (G,B), (G,R), (B,G), (B,R)},
- …
- } (obvious point: *C* is often represented as a function)
- • How did we solve this?

CSP Examle

Variables

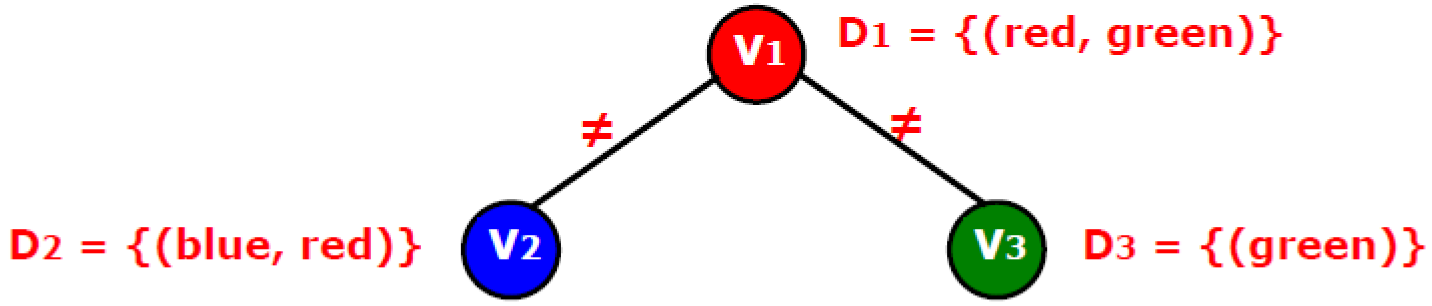
- V1 , V 2, V 3, . . with Domains D1 , D2 , D3 , . . .

Constraints

- Set of allowed value pairs {(red, blue), (green, blue), (green, red)}
- V1 "not equal to" V2,

Solution

- Assign values to variables that satisfy all constraints
- V1 = red , V2 = blue , V3 = green ,

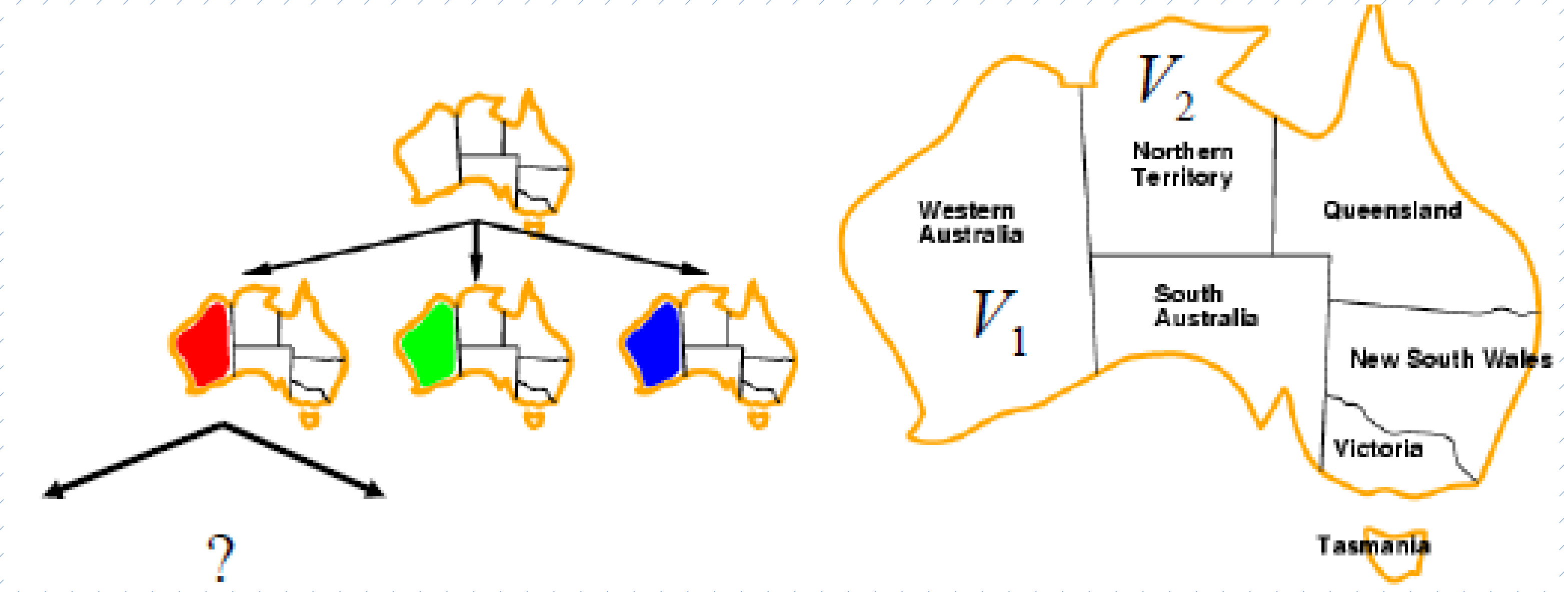


Old solution #2: BFS, DFS, …

- State: partial assignment. (*V*1…*V*k-1 assigned, *V*k…*V*n not yet).
- Start state: all variables unassigned
- Goal state: all assigned, constraints satisfied
- Successor of (*V*1…*V*k-1 assigned, *V*k…*V*n not yet): assign Vk with a value from *D*k
- Cost on transitions: 0 is fine. We don’t care. We just want any solution

Map coloring example

• State: partial assignment. (*V*1…*V*k-1 assigned, *V*k…*V*n not yet).
• Start state: all variables unassigned
• Goal state: all assigned, constraints satisfied
• Successor of (*V*1…*V*k-1 assigned, *V*k…*V*n not yet): assign Vk with a value from *D*k
• Cost on transitions: 0 is fine. We don't care. We just want any solution



It turns out BFS is bad. Why?

Representation of CSP

A CSP is usually represented as an undirected graph, called **Constraint Graph** where the nodes are the variables and the edges are the binary constraints. Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyperarcs.

Solving CSPs

There have four popular solution methods for CSPs amely, *Generate-and-Test*, *Backtracking*, *Consistency Driven, and Forward Checking*.

Generate and Test
We generate one by one all possible complete variable assignments and for each we test if it satisfies all constraints. The corresponding program structure is very simple, just nested loops, one per variable. In the innermost loop we test each constraint. In most situation this method is intolerably slow.

Backtracking
We order the variables in some fashion, trying to place first the variables that are more highly constrained or with smaller ranges. This order has a great impact on the efficiency of solution algorithms and is examined elsewhere. We start assigning values to variables. We check constraint satisfaction at the earliest possible time and extend an assignment if the constraints involving the currently bound variables are satisfied.

Solving CSPs

Consistency Driven Techniques

Consistency techniques effectively rule out many inconsistent labeling at a very early stage, and thus cut short the search for consistent labeling. These techniques have since proved to be effective on a wide variety of hard search problems. The consistency techniques are deterministic, as opposed to the search which is non-deterministic. Thus the deterministic computation is performed as soon as possible and non-deterministic computation during search is used only when there is no more propagation to done. Nevertheless, the consistency techniques are rarely used alone to solve constraint satisfaction problem completely (but they could).

In binary CSPs, various consistency techniques for constraint graphs were introduced to prune the search space. The consistency-enforcing algorithm makes any partial solution of a small subnetwork extensible to some surrounding network. Thus, the potential inconsistency is detected as soon as possible.

Node Consistency, Arc Consistency , Path Consistency (K-Consistency)

Forward Checking

Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency to the instantiated variables, it performs restricted form of arc consistency to the not yet instantiated variables. We speak about restricted arc consistency because forward checking checks only the constraints between the current variable and the future variables. When a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

Confession

- It is possible that some sentences or some information were included in these slides without mentioning exact references. I am sorry for violating rules of intellectual property. When I will have a bit more time, I will try my best to avoid such things.
- These slides are only for students in order to give them very basic concepts about the giant, “Networking”, not for experts.
- Since I am not a network expert, these slides could have wrong/inconsistent information…I am sorry for that.
- Students are requested to check references and Books, or to talk to Network engineers.