

# CSCI 3230

## Fundamentals of Artificial Intelligence

Chapter 4 (Sects.3.5–3.6, 4.1)

### INFORMED AND STOCHASTIC SEARCH ALGORITHMS

# Outline

- ▶ Best-first search (3: uniform cost, greedy & A\*)
- ▶ A\* search
- ▶ Heuristics & Memory-bounded Heuristic Search
- ▶ Hill-climbing (3 iterative improvement)
- ▶ Simulated annealing
- ▶ Genetic Algorithms

# Review: Tree search

```
function Tree-Search(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  Insert(Make-Node(Initial-State[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  Remove-Front(fringe)
    if Goal-Test[problem] applied to State(node) succeeds return node
    fringe  $\leftarrow$  Insert-All(Expand(node, problem), fringe)
```

A strategy is defined by picking **the order of node expansion**  
Insert-All in the fringe (queue) in the order accordingly

# Best-first search

- ▶ **Evaluation function  $f$**  – the desirability of expanding node. Nodes ordered so the **best** evaluation is **expanded first**. The strategy is called **best-first** search.
- ▶ If the evaluation function,  $f$ , is **omniscient**, then this will indeed be the best node. In reality,  $f$  will sometimes be off, and can lead the search astray.
- ▶ Measure seen: **path cost  $g$**  (in **uniform-cost** search in Ch3) to decide which path to expand. Does **not direct** search toward the goal. To focus/direct the search, the measure must **estimate** the cost of the path from current state to the closest goal state.

# Greedy search (Best First = BF)

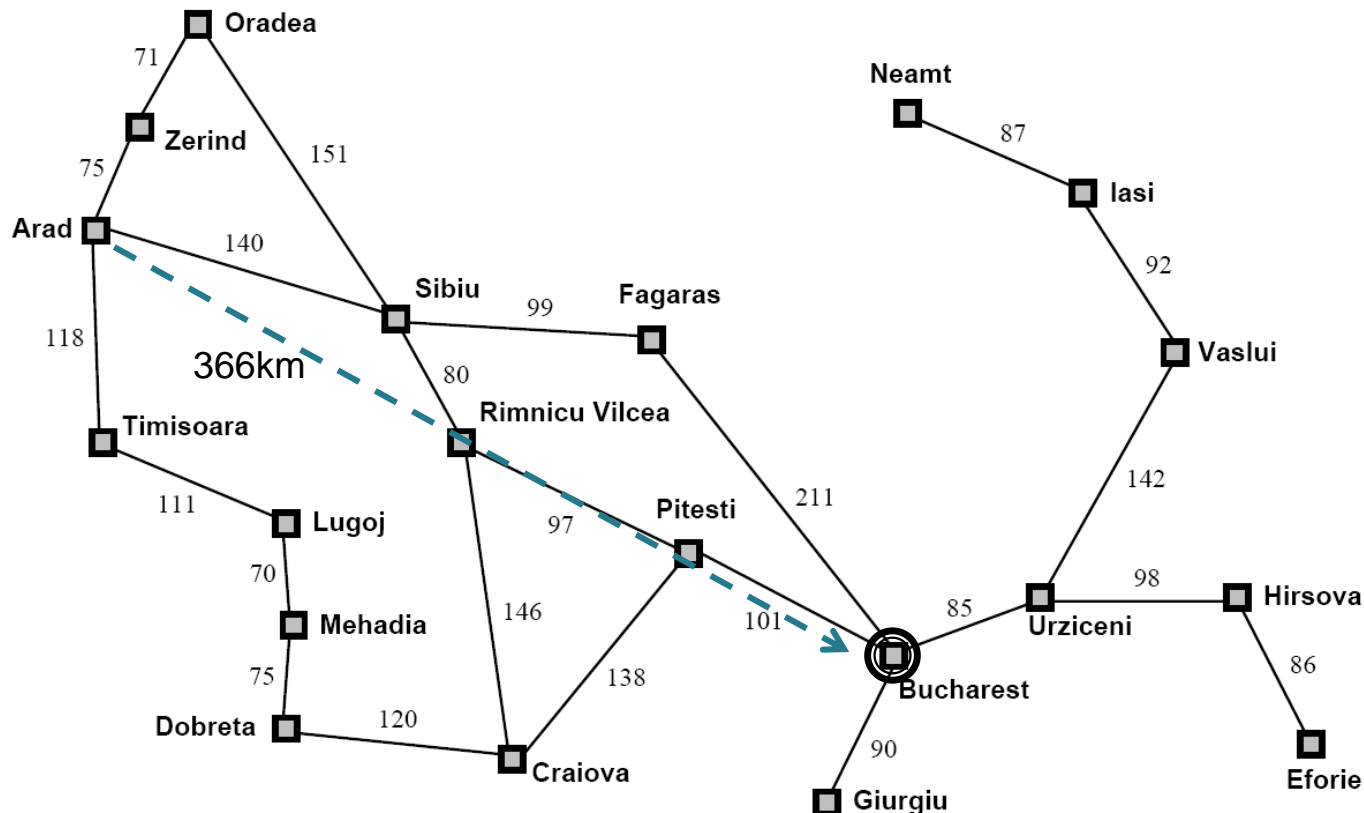
One of the simplest best-first strategies is to **minimize** the estimated cost to reach the goal.

**Evaluation function** = estimate of cost from  $n$  to the closest goal,  $h(n)$  (heuristic  $fn$ )

E.g.  $h_{SLD}(n)$  = straight-line distance from  $n$  to Bucharest

- ▶ Formally speaking,  $h$  can be any function at all, as long as  $h(n) = 0$  if  $n$  is a goal.
- ▶ A BF search that uses  $h$  to select the next node to expand is called **greedy search**.
- ▶ Greedy search expands the node that **appears** to be closest to goal
- ▶ Takes the **biggest bite** possible out of the remaining cost to the goal, not **global** optimal – hence the name “greedy search”.

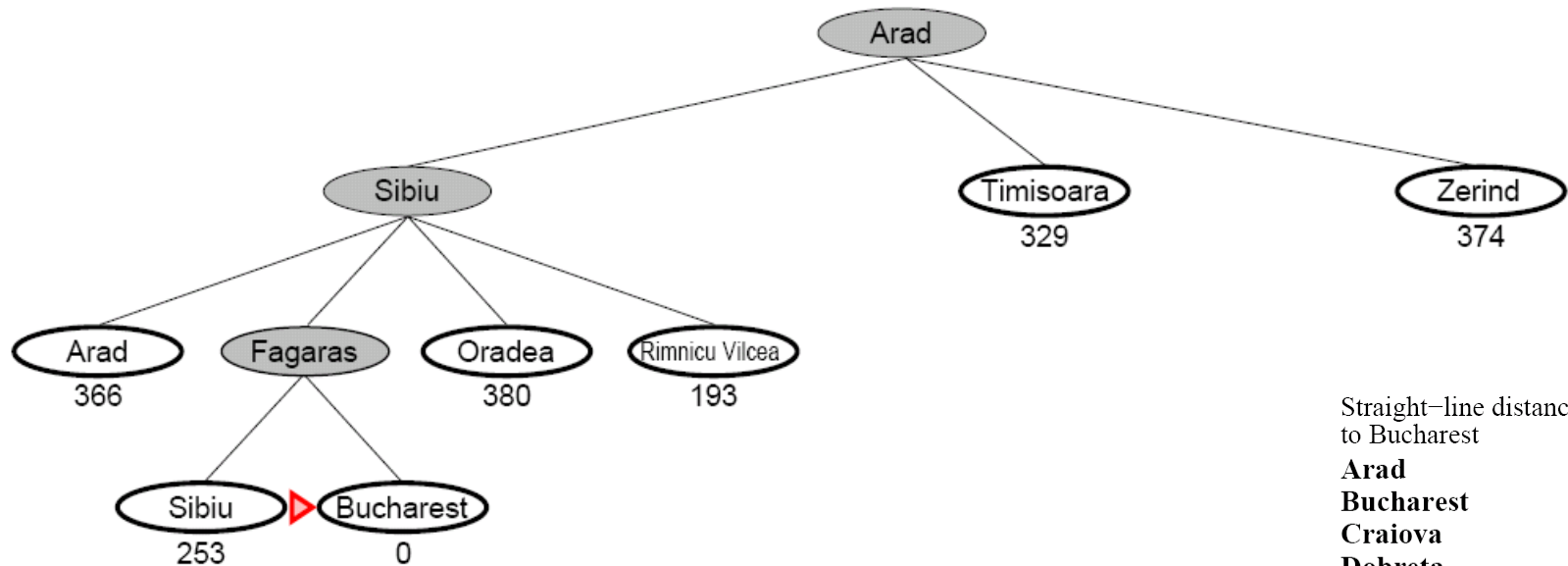
# Romania with step costs in km



Straight-line distance  
to Bucharest

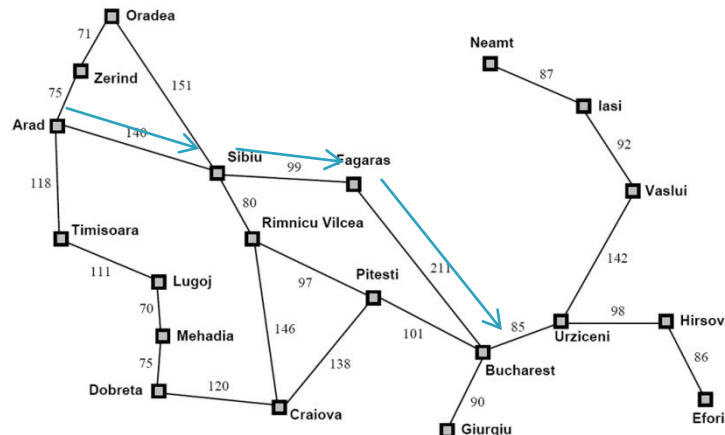
<b>Arad</b>	366
<b>Bucharest</b>	0
<b>Craiova</b>	160
<b>Dobreta</b>	242
<b>Eforie</b>	161
<b>Fagaras</b>	176
<b>Giurgiu</b>	77
<b>Hirsova</b>	151
<b>Iasi</b>	226
<b>Lugoj</b>	244
<b>Mehadia</b>	241
<b>Neamt</b>	234
<b>Oradea</b>	380
<b>Pitesti</b>	100
<b>Rimnicu Vilcea</b>	193
<b>Sibiu</b>	253
<b>Timisoara</b>	329
<b>Urziceni</b>	80
<b>Vaslui</b>	199
<b>Zerind</b>	374

# Greedy search example



Straight-line distance  
to Bucharest

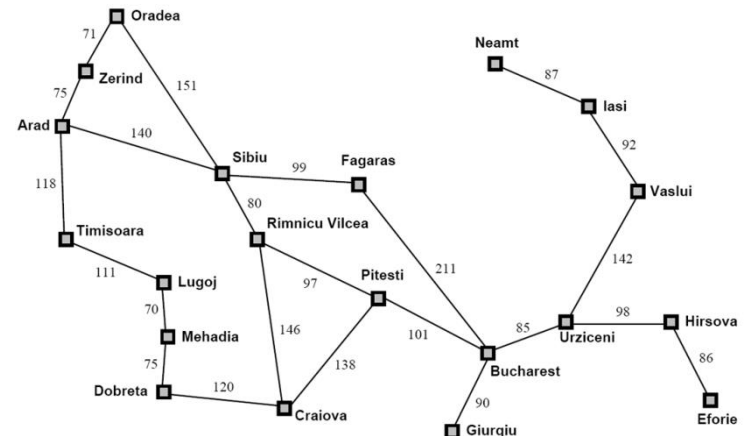
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



# Properties of greedy search

Complete	No, – can get stuck in loops, e.g. from Iasi to Fagaras Iasi (ya-sh) → Neamt → Iasi → Neamt → ... Complete in finite space with repeated-state checking
Time	$O(b^m)$ : but a good heuristic can give dramatic improvement, $m$ : max depth of search tree
Space	$O(b^m)$ – keeps all nodes in memory
Optimal	No

?loops in algorithms using path value  $g$ ?





# A\* search

Idea: avoid expanding paths that are already expensive

Evaluation function  $f(n) = g(n) + h(n)$

$g(n)$  = cost so far to reach  $n$

$h(n)$  = estimated cost to goal from  $n$

$f(n)$  = estimated total cost of path through  $n$  to goal

A\* search uses an **admissible** heuristic

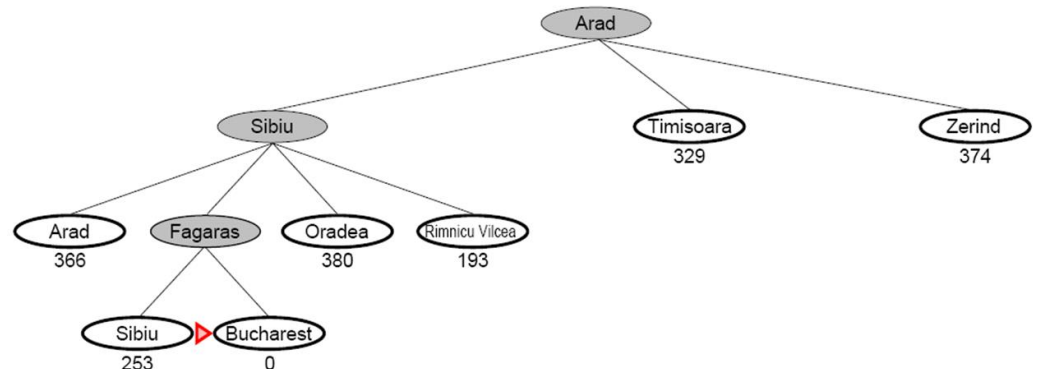
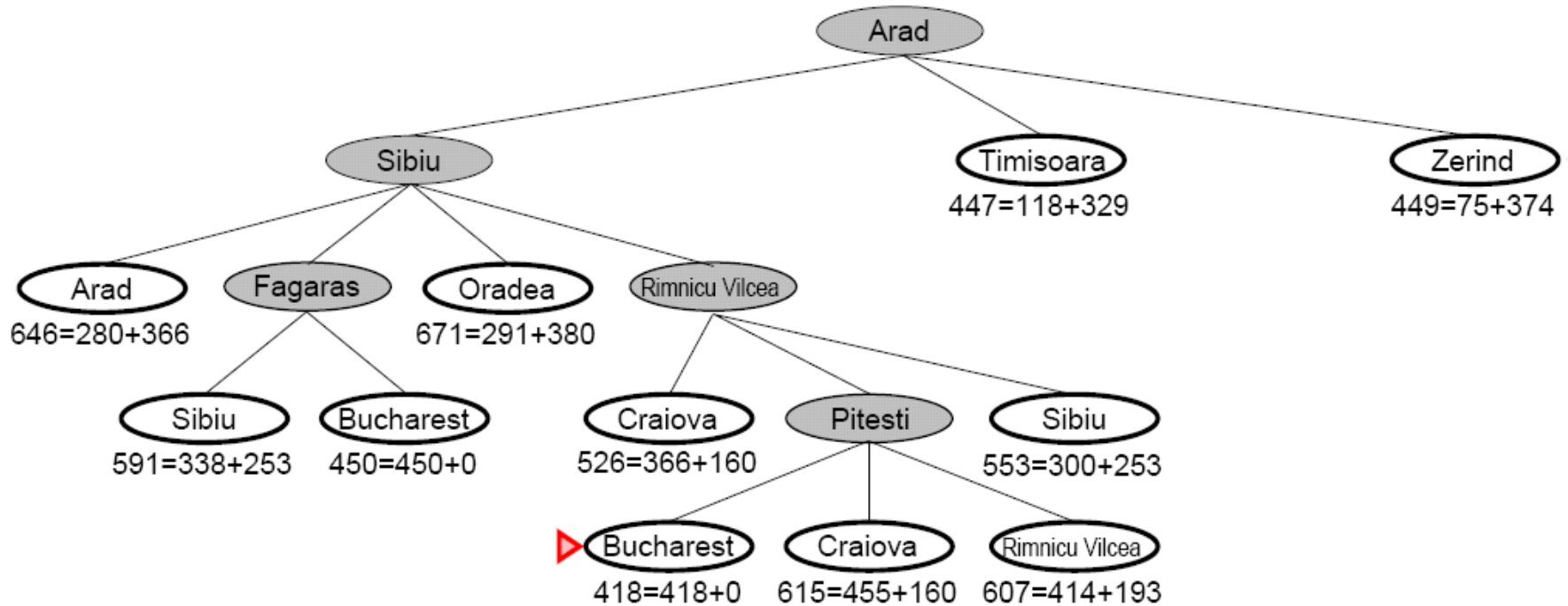
i.e.  $h(n) \leq h^*(n)$  where  $h^*(n)$  is the **true** cost from  $n$

(Also require  $h(n) \geq 0$ , so  $h(G) = 0$  for any goal  $G$ .)  $f(n) \leq f^*(n) = g^*(G)$

E.g.  $h_{SLD}(n)$  never overestimates the actual road distance

**Theorem:** A\* search is optimal.

# A\* search example



# Optimality of A\* (proof by contradiction)

Let  $G$ : optimal goal states with path cost  $f^*$

Let  $G_2$ : suboptimal,  $g(G_2) > f^*$  (i)

Imagine A\* selected  $G_2$  as a goal state

Consider

node  $n$  on an optimal path to  $G$

Because  $h$  is admissible

$$f^* \geq f(n)$$

If  $n$  is not chosen for expansion over  $G_2$ :

$$f(n) \geq f(G_2)$$

$$\Rightarrow f^* \geq f(G_2)$$

Because  $G_2$  is a goal state,  $\Rightarrow h(G_2) = 0$

$$\Rightarrow f(G_2) = g(G_2)$$

$$\Rightarrow f^* \geq g(G_2)$$

$\Rightarrow$  contradicts with (i) and A\* will never select a suboptimal goal ( $G_2$ ) for expansion

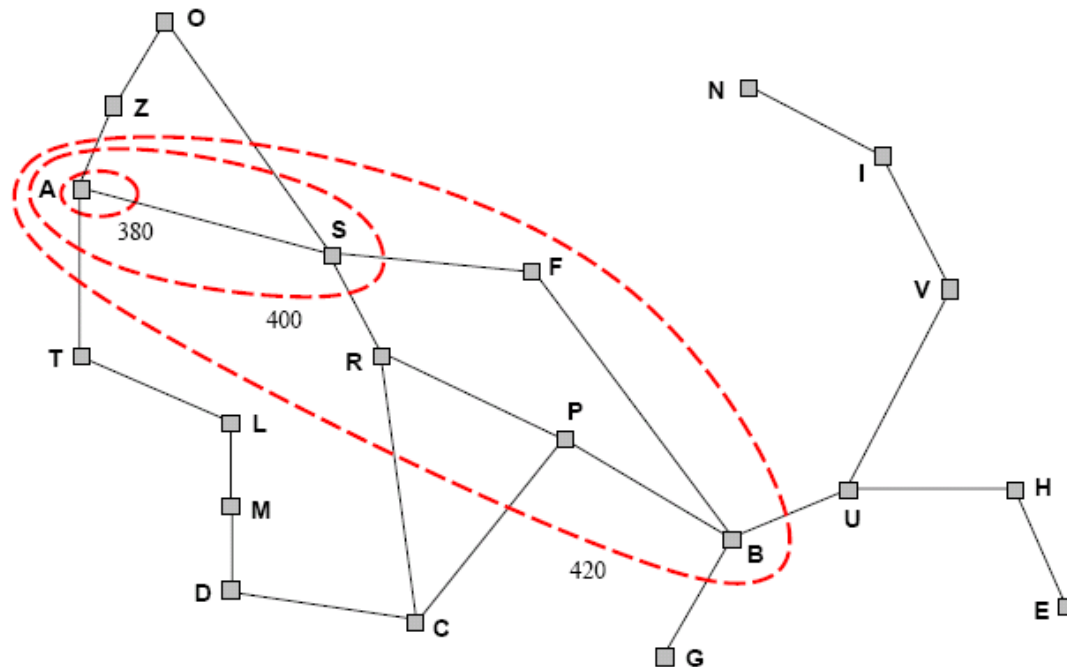
$\Rightarrow$  A\* is optimal.

# Optimality of A\* (more useful)

Lemma: A\* expands nodes in order of increasing  $f$  value\*. (proof on p.14)

Gradually adds “ $f$ -contours” of nodes (cf. breadth-first adds layers)

Contour  $i$  has all nodes with  $f = f_i$ , where  $f_i < f_{i+1}$



?Use  $g$  only

?Use  $h$  only

Intuitively, 1<sup>st</sup> goal the contours touch expanding outward must be optimal because anything outside  $> f(G) = g(G)$ , ( $h(G) = 0$ )

$$f_i \leq f(G) = g(G)$$

# Proof of lemma: Consistency

Lemma: A\* expands nodes in order of increasing  $f$  value\*.

A heuristic is **consistent** if

$$h(n) \leq c(n, a, n') + h(n')$$

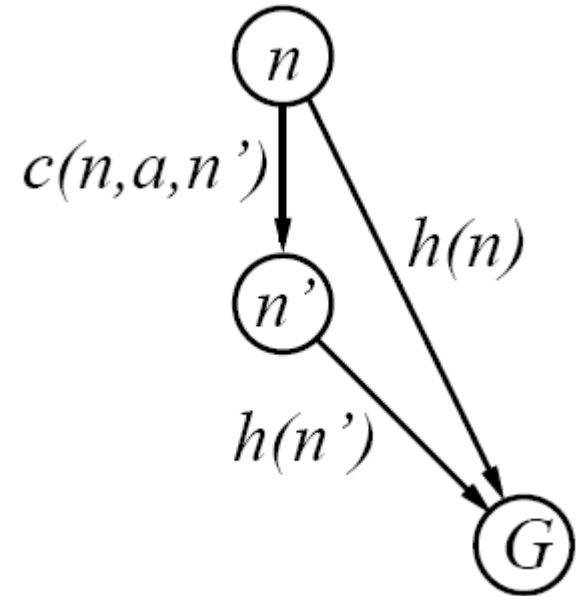
If  $h$  is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

$$f(n') \geq f(n)$$

i.e.  $f(n)$  is non-decreasing along any path.

$c(n, a, n')$  must  $\geq 0$ , (step cost +ve or 0)



# Properties of A\*

Complete	Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time	Exponential in [relative error in $h$ x length of solution]
Space	Keeps all nodes in memory
Optimal	Yes – cannot expand $f_{i+1}$ until $f_i$ is finished

The condition for sub-exponential growth is that

$$|h(n) - h^*(n)| \leq O(\log h^*(n)).$$

where the 1<sup>st</sup> term is error in  $h$  and  $h^*(n)$  is actual cost from  $n$  to  $G$ .

# Optimal Efficiency of A\*

- A\* expands all nodes with  $f(n) < C^*$  (*optimal path cost*)
  - A\* expands some nodes with  $f(n) = C^*$
  - A\* expands no nodes with  $f(n) > C^*$
- ⇒ A\* is **optimally efficient** for any given heuristic function.

- That is, no other optimal algorithm is guaranteed to expand fewer nodes than A\*.
- Any algorithm that does NOT expand all the nodes with  $f(n) < C^*$  runs the risk of missing the optimal solution.

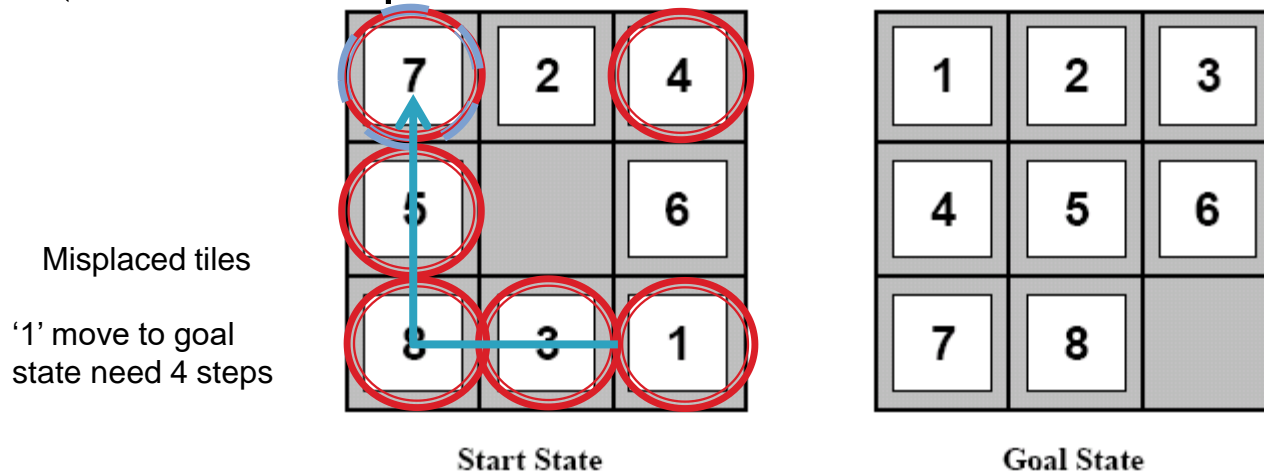
# Admissible heuristics, A\*

E.g. for the 8-puzzle

$h_1(n)$  = number of misplaced tiles

$h_2(n)$  = total **Manhattan** distance

(i.e. no. of square from desired location of each tile)



$$h_1(S) = 6$$

$$h_2(S) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$$



# Dominance, A\*

If  $h_2(n) \geq h_1(n)$  for all  $n$  (both admissible)  
then  $h_2$  **dominates**  $h_1$  and is better (more efficient) for search

## The effect of heuristic accuracy on performance:

- ▶ To characterize the quality of a heuristic: the **effective branching factor**  $b^*$ .
- ▶ Total number of nodes expanded by A\*:  $N$ , and the solution depth:  $d$
- ▶ A uniform tree of depth  $d$

$$N = 1 + b^* + (b^*)^2 + \dots + (b^*)^d.$$

e.g. if A\* finds a solution at depth 5 using 52 (N) nodes, then  $b^*$  is 1.91 ?

- ▶ Usually,  $b^*$  for a given heuristic is fairly **constant** over a large range of problem instances, and therefore **experimental** measurements of  $b^*$  on a small set of problems can provide a good guide to the heuristic's overall usefulness.

# Effect of heuristic accuracy on performance

	Search Cost			Effective Branching Factor $b^*$		
$d$	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	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	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

Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A\* algorithms with  $h_1$  and  $h_2$ . Data averaged over 100 instance of the **8-puzzle** for various solution lengths.

# Inventing admissible $h$ by Relaxed problems

Admissible heuristics can be derived from the **exact** solution cost of a **relaxed** version of the problem.

If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then  $h_1(n)$  gives the shortest solution.

If the rules are relaxed so that a tile can move to **any adjacent square**, then  $h_2(n)$  gives the shortest solution.

**Key point:** the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem **admissible**

7	2	4
5		6
8	3	1

Start State

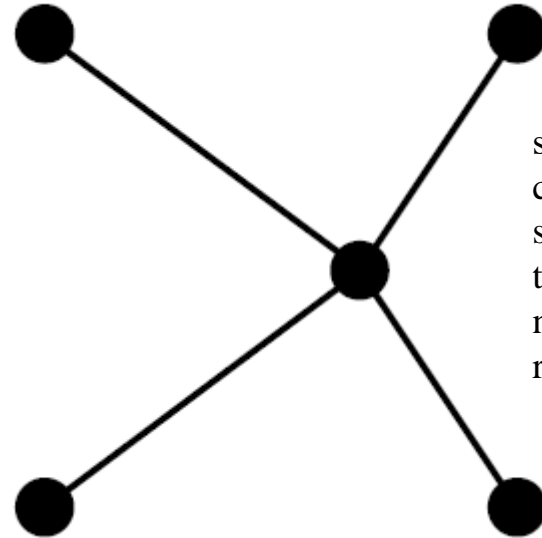
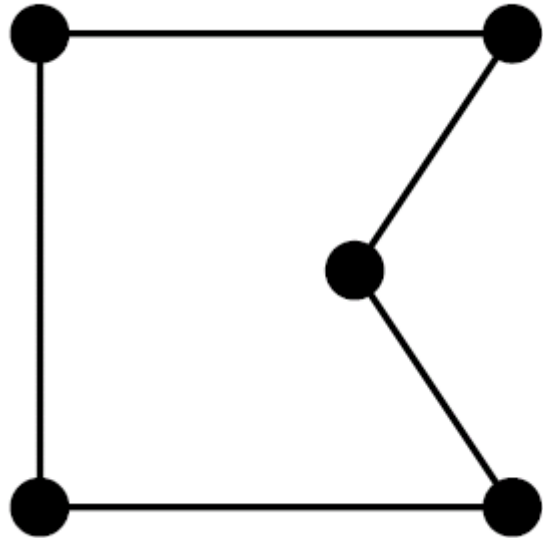
1	2	3
4	5	6
7	8	

Goal State

# Relaxed problems

**Well-known example:** travelling salesperson problem (TSP)

- Find the shortest tour visiting all cities exactly once



spanning tree's  
centre is the  
starting node of  
the remaining  
nodes of the  
remaining tour

**Minimum spanning tree** can be computed in  $O(n^2)$  and is a lower bound on the shortest tour

# Memory-bounded Heuristic Search

To reduce the memory requirement by trading off with repeated search:

- Iterative-deepening  $A^*$  (**IDA\***)
- Recursive best-first search (**RBFS**); memory  $O(bd)$
- Memory-bounded  $A^*$  (**MA\***)
- Simplified  $MA^*$  (**SMA\***)

The last 2 maximize the use of available memory

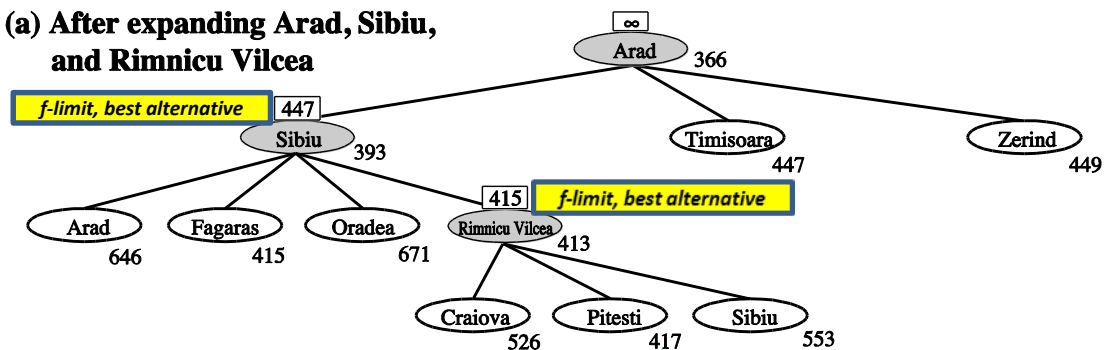
# Recursive Best First Search (RBFS)

Memory complexity:  $O(bd)$   
 ?Why

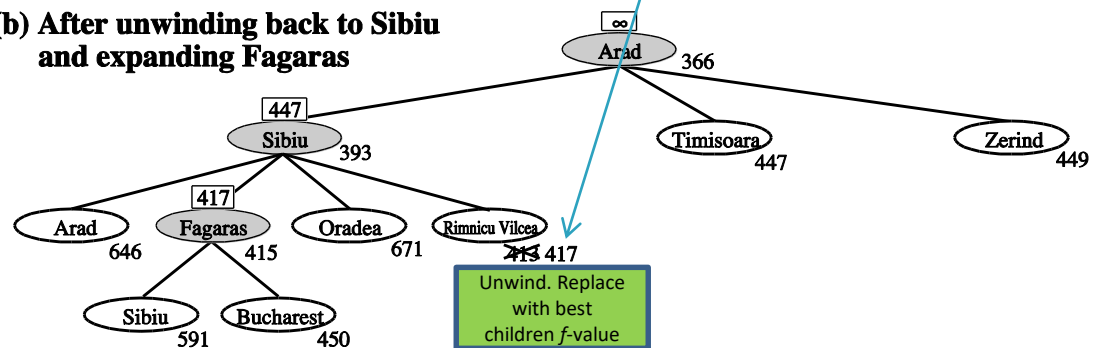
Only  $bd$  nodes kept anytime!

Always one expanded node at one depth

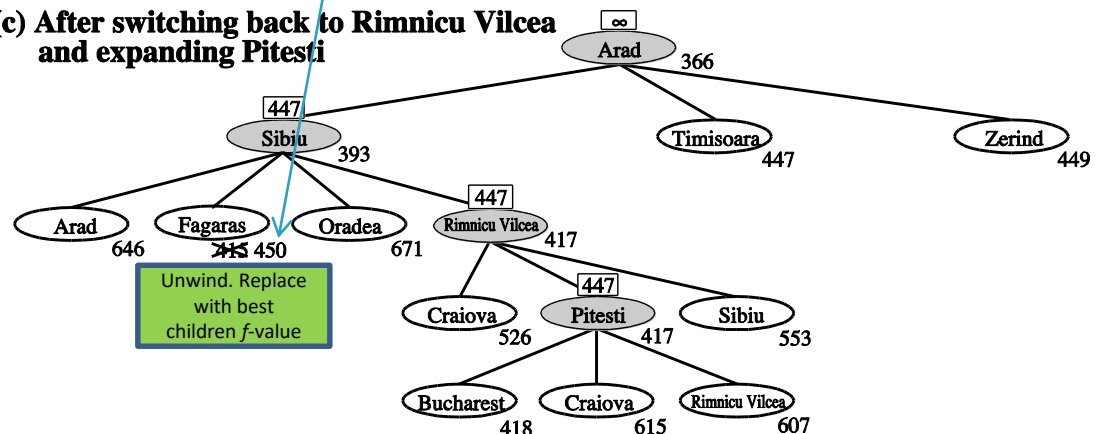
(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti

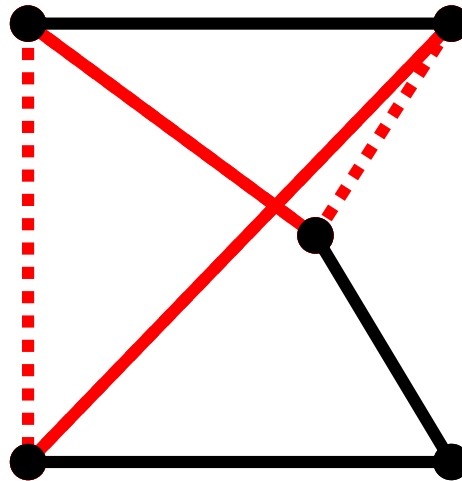


# Iterative improvement algorithms

- ▶ In many optimization problems, **path** is irrelevant (not part of the solution); the goal state itself is the solution. Does the path affect the efficiency or chance of finding the global optimal ??
- ▶ State space = set of “complete” configurations;  
find **optimal** configuration, e.g. TSP  
or, find configuration satisfying constraints, e.g., timetable
- ▶ In such cases, can use **iterative improvement** algorithms; keep a single “current” state, try to improve it; **types of search??**
- ▶ Constant space, suitable for online/offline search

# Example: Travelling Salesperson Problem

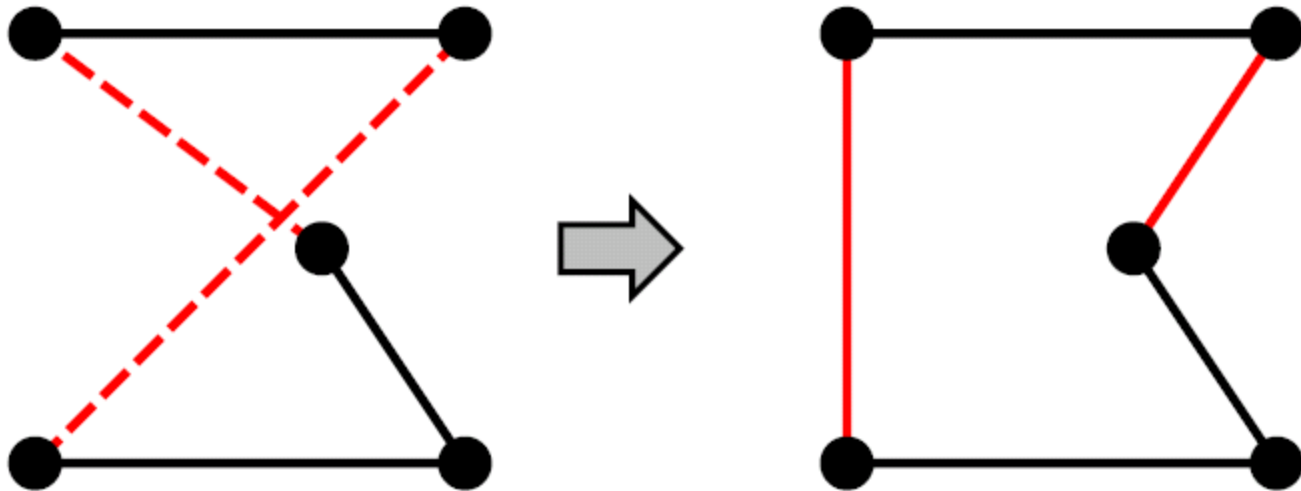
Start with any complete tour, perform pair-wise exchanges





# Example: Travelling Salesperson Problem

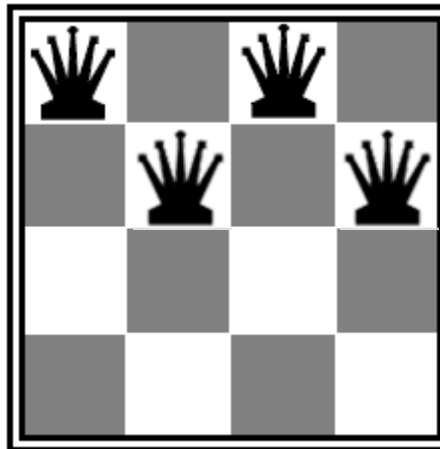
Start with any complete tour, perform pair-wise exchanges



## Example: $n$ -queens

Rules: Put  $n$  queens on an  $n \times n$  board with no two queens on the same row, column or diagonal

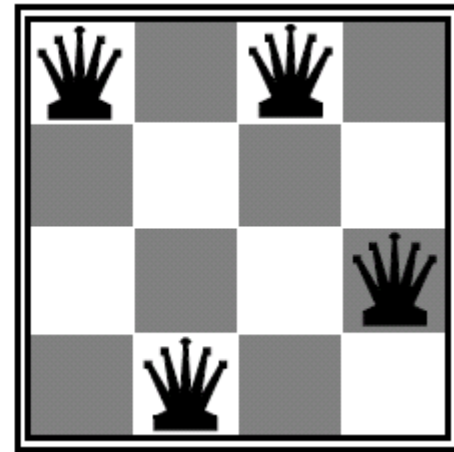
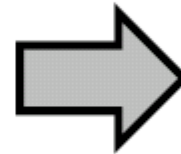
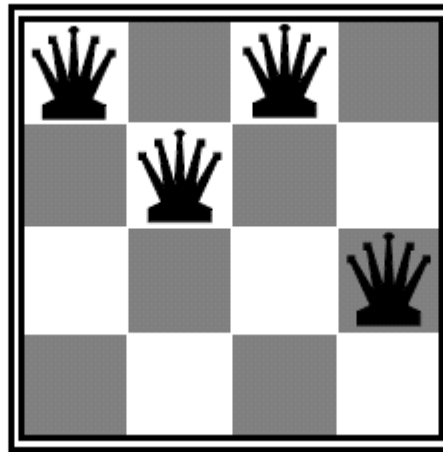
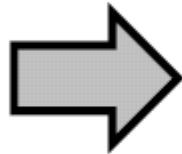
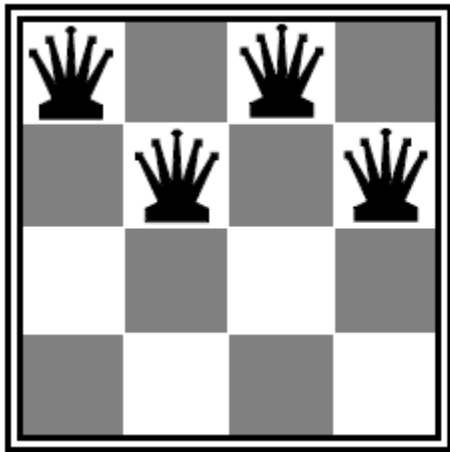
- ▶ Move a queen to reduce number of conflicts



## Example: $n$ -queens

Rules: Put  $n$  queens on an  $n \times n$  board with no two queens on the same row, column or diagonal

- ▶ Move a queen to reduce number of conflicts



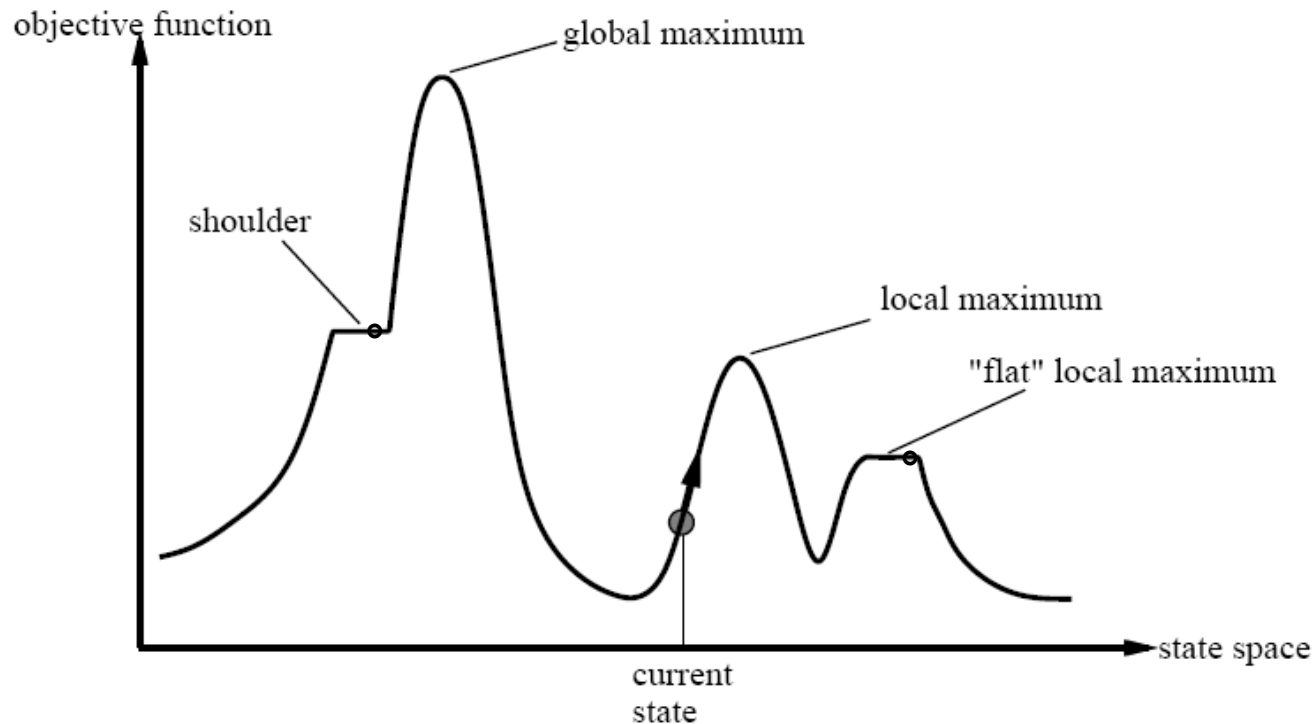
# Hill Climbing (or gradient ascent / descent)

“Like climbing Everest in thick fog with amnesia”

```
function Hill-Climbing (problem) returns a state that is a local maximum
  inputs:           problem , a problem
  local variables: current, a node
                   neighbor, a node
  current  $\leftarrow$  Make-Node ( Initial-State[problem])
  loop do
    neighbor  $\leftarrow$  a highest-valued successor of current
    if Value[neighbor] < Value[current] then return State[current]
    current  $\leftarrow$  neighbor
  end
```

# Hill-climbing

Problem: depending on initial state, can get stuck on local maxima ?



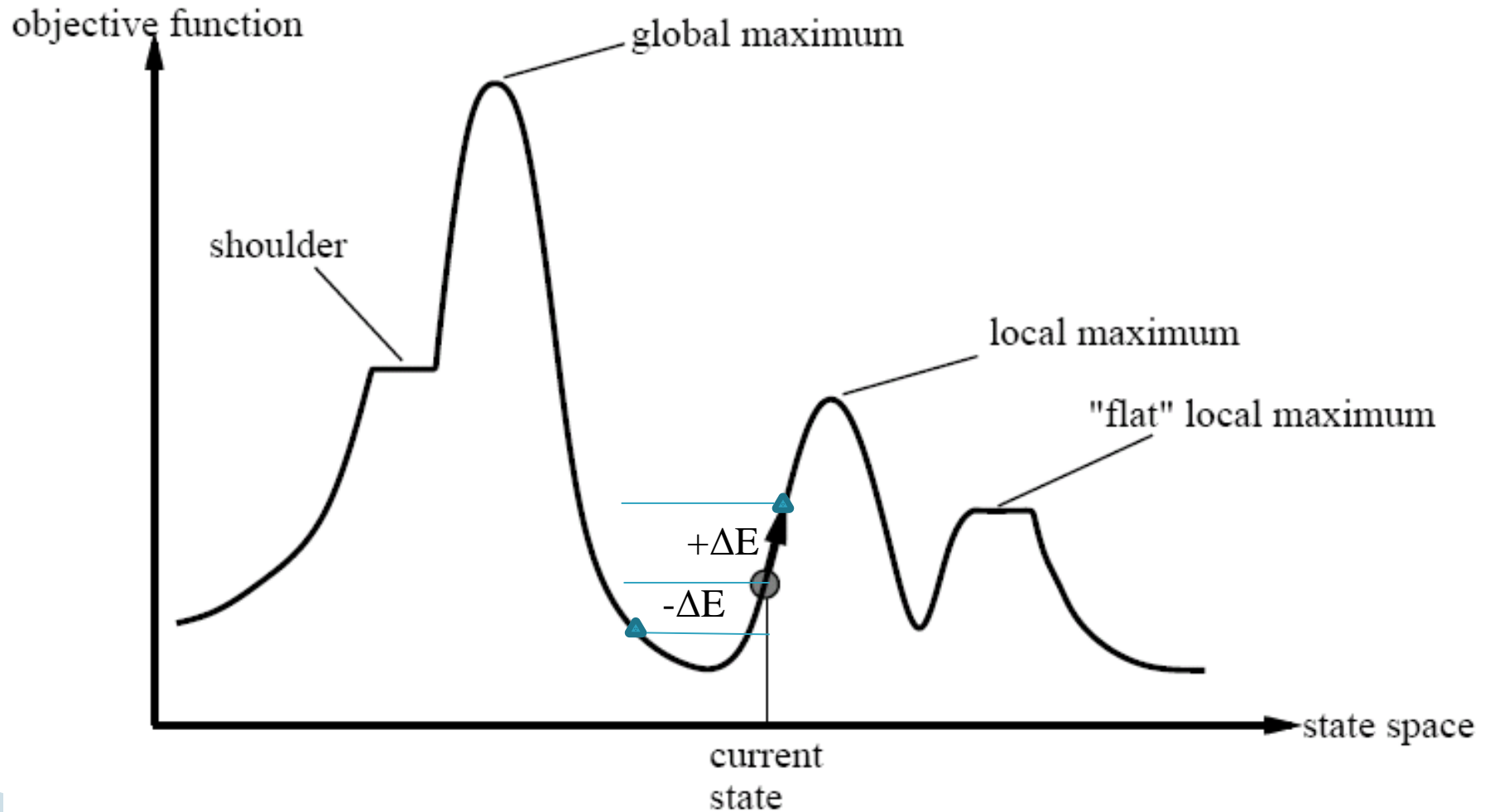
In continuous spaces, problems with choosing step size, slow convergence  
Random multi-(re)start? Discrete?? Adaptive? Escape? Discuss!

# Simulated annealing

Idea: escape local maxima by allowing some “bad” moves **but gradually decrease their size and frequency**

```
function Simulated-Annealing (problem, schedule) returns a solution state //maximization
  inputs: problem, a problem
             schedule, a mapping from time  $t$  to “temperature”
  local variables: current, a node
                     next, a node
                      $T$ , a “temperature” controlling probability of downward steps
  current  $\leftarrow$  Make-Node(Initial-State[problem])
  for  $t \leftarrow 1$  to  $\infty$  do
     $T \leftarrow$  schedule[ $t$ ]
    if  $T = 0$  then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  Value[next] – Value[current]
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 
```

# Simulated annealing



# Properties of simulated annealing $e^{\Delta E/T}$

- ▶ Instead of starting again randomly when stuck on a local maximum, we could allow the search to take some **downhill** steps to **escape** the local maximum.
- ▶ Instead of picking the best move, however, it picks a **random** move.
- ▶ If the move actually **improves** the situation, it is **always executed**. Otherwise, the algorithm makes the move with some **probability**  $< 1$ . The probability decreases exponentially with the “badness” of the move: the amount  $\Delta E$  (–ve) by which the evaluation is worsened.
- ▶ A second parameter  **$T$**  is also used to determine the probability. At higher values of  $T$ , “bad” moves are more likely to be allowed. As  $T$  tends to zero, they become more and more unlikely, until the algorithm behaves more or less like **hill-climbing**.
- ▶ The ***schedule*** input determines the value of  $T$  as a function of how many cycles already have been completed.  $T \leftarrow \text{schedule}[t]$



## Properties of simulated annealing 2 $e^{\Delta E/T}$

- ▶ The Value function ( $\rightarrow \Delta E$ ) corresponds to the total **energy** of the atoms in the material, and **T** corresponds to the **temperature**. The **schedule** determines the rate at which the temperature is lowered.
- ▶ It can be proved that if *schedule* lowers  $T$  slowly enough, SA can find a global optimum.
- ▶ T decreased slowly enough  $\Rightarrow$  always reach best state  
**Is this necessarily an interest guarantee??**
- Devised by Metropolis et al, 1953, for physical process modeling; widely used in VLSI layout, airline scheduling. Etc.

Adaptive SA? Schedule, random step size

Demo->

# Genetic algorithms EC: GA, GP, EP, ES

- ▶ **Genetic algorithm (GA):** stochastic group search for global optimum (optima).
- ▶ **Initial population:** a set <sup>k=population size</sup> k randomly generated states (**individuals** or **chromosomes**) which evolve to search for the optima through genetic operations:
- ▶ **Crossover**
- ▶ **Mutation**
- ▶ **Selection**
- ▶ **Fitness (evaluate) function** to compute the individual's fitness used in the proportionate selection

Local operators; memetic approach

# Genetic Algorithm

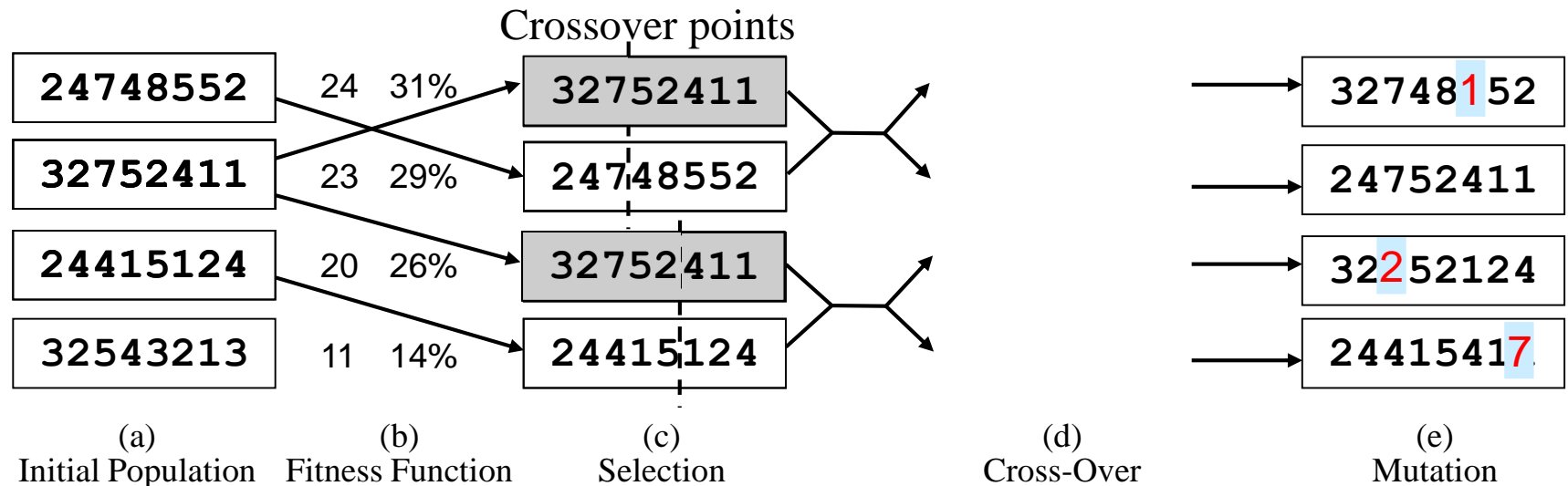
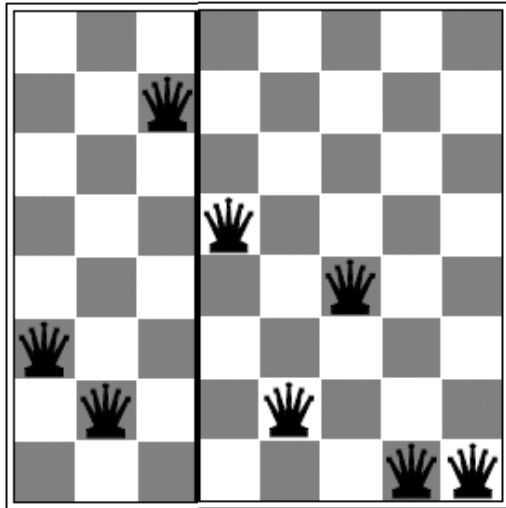


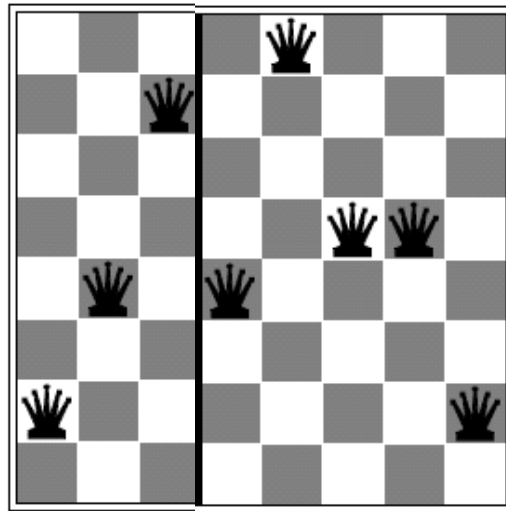
Fig 4.15 The GA. The initial population in (a) is ranked by the fitness function in (b). Resulting the pairs of mating in (c). They produce offspring in (d), which are subject to mutation in (e).

# Genetic Algorithm



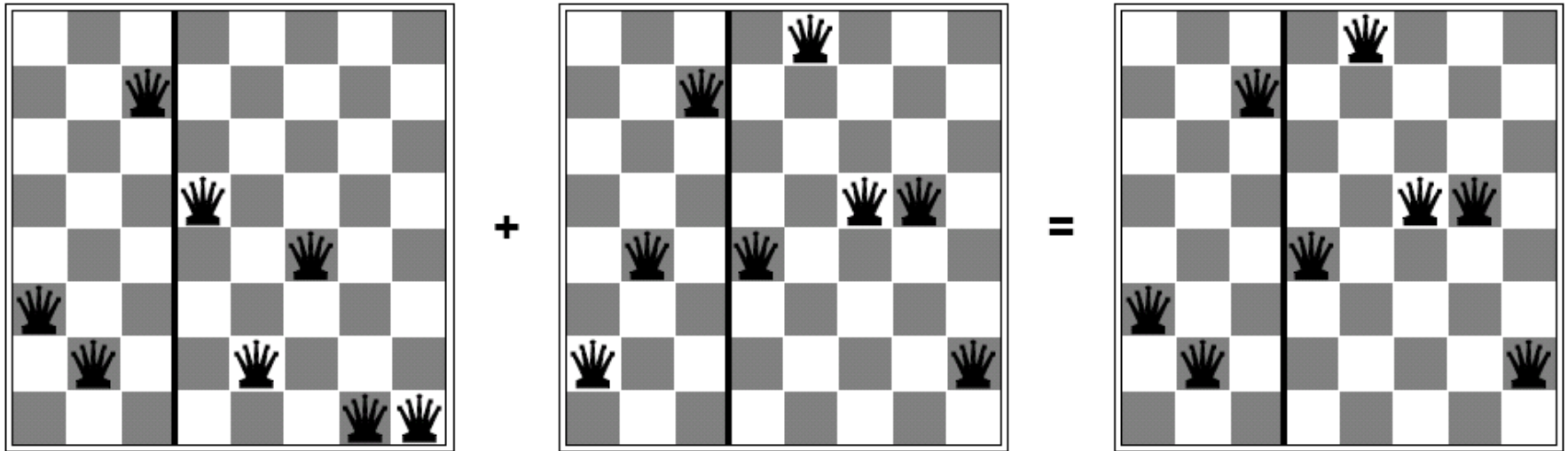
+

=



2 parents in Fig 4.15(c)  
and 1<sup>st</sup> offspring in Fig 4.15 (d)

# Genetic Algorithm



2 parents in Fig 4.15(c) and 1<sup>st</sup> offspring in Fig 4.15 (d)

# Genetic Algorithm

Genetic Algorithm Fish [YouTube]

<http://metivity.com>

Evolution of fish. Each fish is a **neural net**.

**Objective:** develop a "killer fish" - a fish that eats maximum pieces of food in a constant time period.

The world is populated by 20 fish and 40 pieces of food.

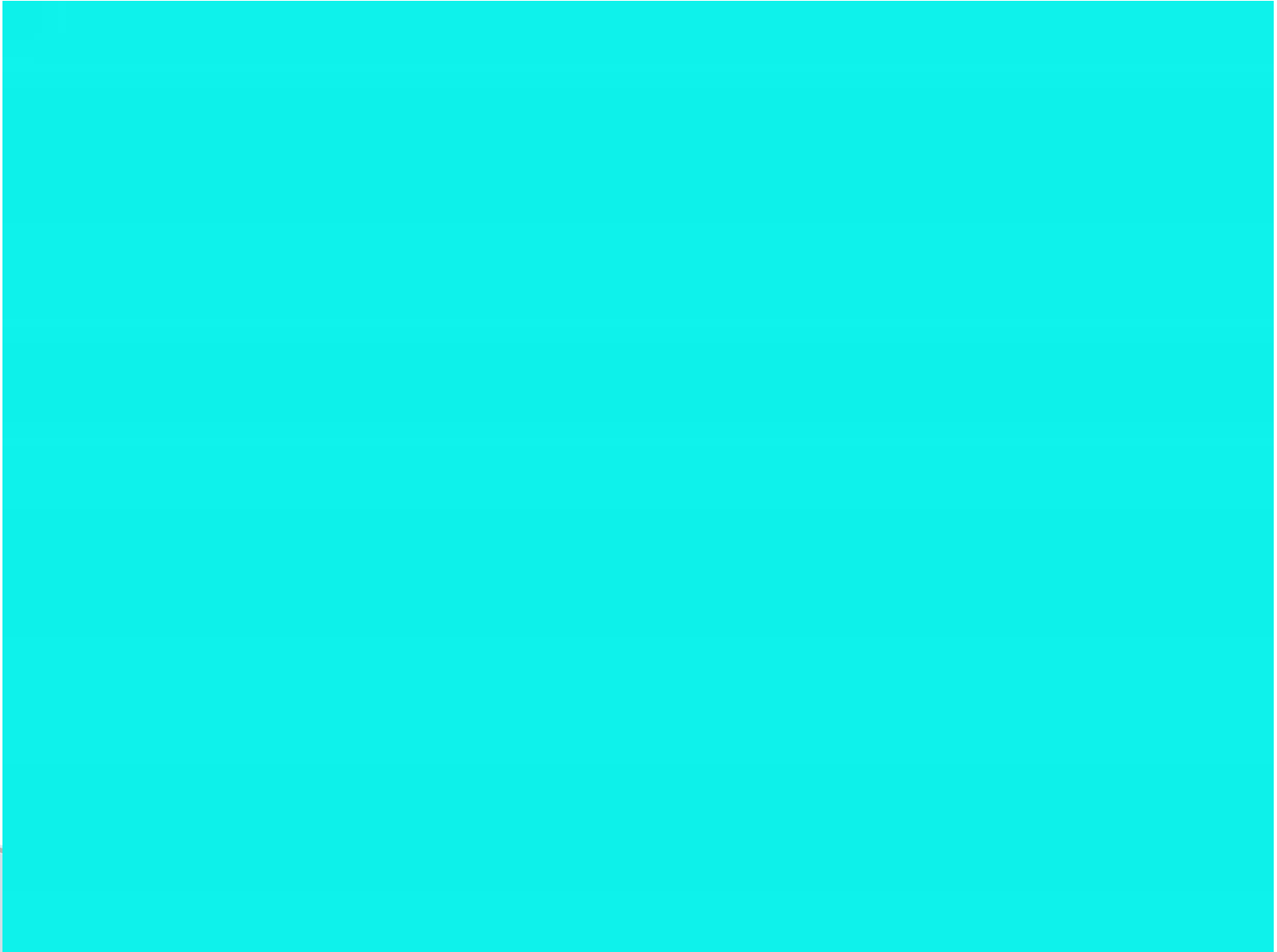
Every time a generation begins, the food is scattered in a random distribution on a certain random location on the screen. (random piles of food).

When being eaten, the food shows up in a new random pile in the screen.

Look how they improve as time goes by. Their logic is being build by evolution, without interference.

# Genetic Algorithm

[Genetic Algorithm Fish](#) [YouTube]



**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

**repeats**

*new\_population*  $\leftarrow$  empty set

**loop for** *i* **from** 1 **to** Size(*population*) **do**

*x*  $\leftarrow$  Random-Selection(*population*, *Fitness-Fn*) // fitness proportionate

*y*  $\leftarrow$  Random-Selection(*population*, *Fitness-Fn*)

*child*  $\leftarrow$  Reproduce(*x*,*y*) // crossover

**if** (small random probability) **then** *child*  $\leftarrow$  Mutate(*child*) // mutation

add *child* to *new\_population*

*population*  $\leftarrow$  *new\_population*

**until** some individual is fit enough, or enough time has elapsed

**return** the best individual in population, according to *Fitness-Fn*

} one generation

---

**function** Reproduce(*x*,*y*) **returns** an individual // crossover

**inputs:** *x*, *y*, parent individuals

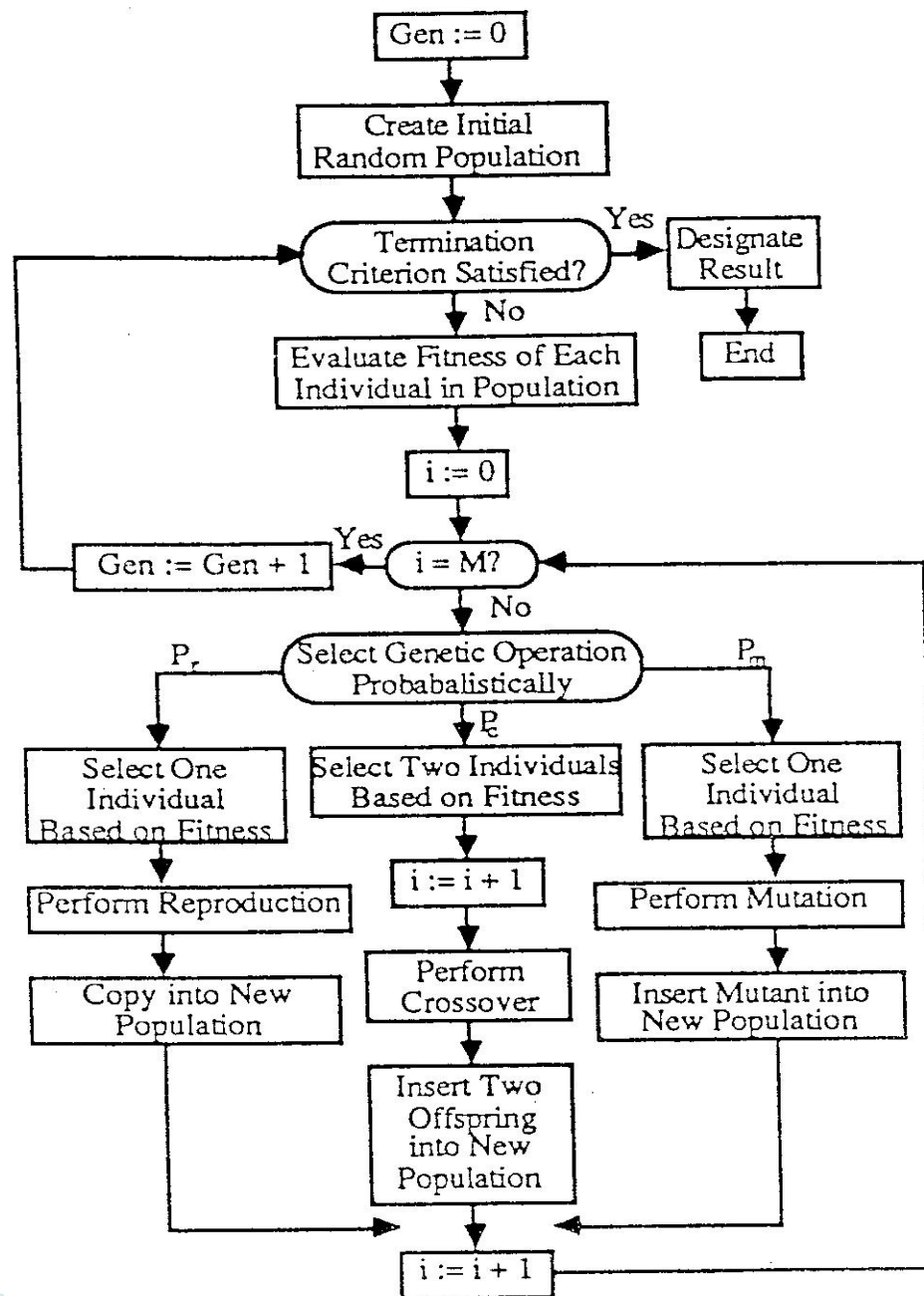
*n*  $\leftarrow$  Length(*x*)

*c*  $\leftarrow$  random number from 1 to *n* // crossover point

**return** Append(Substring(*x*, 1, *c*), Substring(*y*, *c*+1, *n*))

Fig 4.17 A genetic algorithm. The algorithm is the same as the one diagrammed in Fig. 4.15, with one variation: in the more popular version, each mating of two parents produces only one offspring, not two.



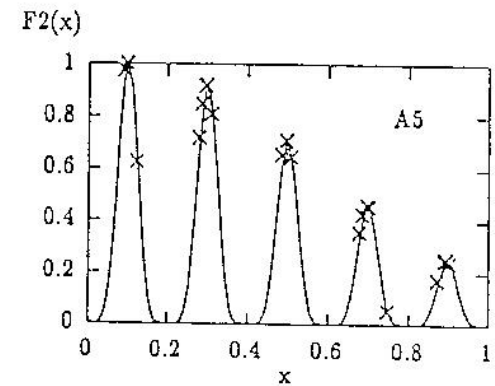


M: population size

P: probability

Figure 3.1 Flowchart of the conventional genetic algorithm.

# Local Maxima are Easier to Handle with Diversity



- ▶ Trap-escaping mechanisms:
  - **Backtracking** and **initially large, ever-shrinking step size** (e.g., **simulated annealing**)
  - Parallel search with a large number of random starting positions (e.g., **multi-start**)
- ▶ Diversity
  - In contrast, if GA treats diversity as a component of fitness, then some individuals tend to hover around already-discovered local maxima in quality or diversity, **driving off** other, still peripatetic, individuals.
  - If there are **enough** individuals to **populate** all the **local** maxima sufficiently, there is a reasonable chance that one individual will find its way to the **global maximum**.

The populate-and-conquer principle:

*Local maxima should be populated, not avoided, when you are seeking a global optimum.*

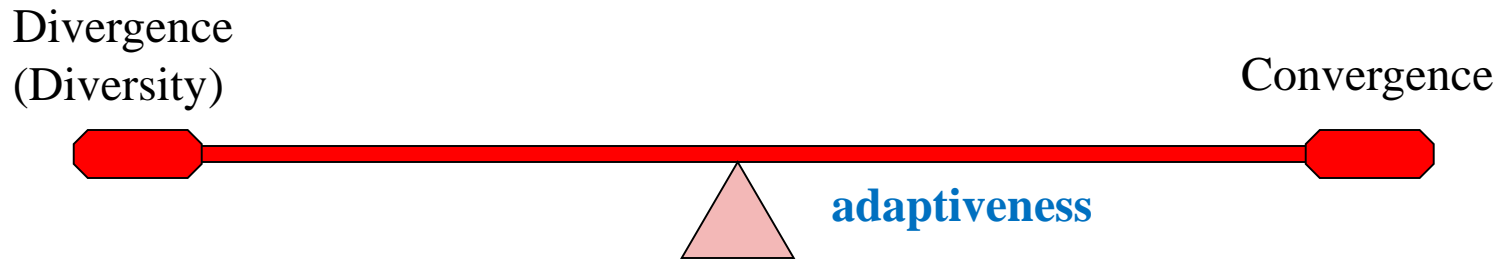
# Genetic Algorithms Generally Involve Many Choices

- ▶ How many chromosomes are to be in the population?
  - Too low: chromosomes soon have identical traits and crossover will do nothing. **Can't cover the search space.** # of individuals vs # of generations
  - Too high: computation time will be unnecessarily excessive
- ▶ What is the mutation rate?/size
  - Too low: new traits will appear too slowly in the population
  - Too high: each generation will be unrelated to the previous one
- ▶ Is mating allowed?
  - How are mating pairs selected?
  - How are crossover points determined? How many points?
- ▶ Can any chromosome appear more than once in the population?
- ▶ Evaluation Methods **(divergence and convergence) e.g.  $1/\text{error}^2$** 
  - Fitness functions (Landscape); regularization.
- ▶ Reproduction?
- ▶ Any novel (local/greedy) operators?  
*Adaptive? Dynamic?*

# Genetic Algorithms Generally Involve Many Choices

- ▶ Generally speaking, it is helpful to know the shape of the problem space to be searched.

**Efficiency & Effectiveness**



(Memetic approach)