CSCI 3230

Fundamentals of Artificial Intelligence

Chapter 4 (Sects.3.5–3.6, 4.1)

隨機的

INFORMED AND STOCHASTIC SEARCH ALGORITHMS

Fall 2013

Outline

- Best-first search (3: uniform cost, <u>greedy & A*</u>)
- A* search
- Heuristics 啟發式的
- Hill-climbing (3 iterative improvement)
- Simulated annealing
- Genetic Algorithms

Review: Tree search

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

A strategy is defined by picking the order of node expansion

Insert in the fringe (queue) in the order accordingly

Best-first search

- Evaluation function – the desirability of expanding node. Nodes are ordered so that the best evaluation is expanded first. The strategy is called best-first search.
- IF the evaluation function, f, is omniscient 全知的, then this will indeed be the best node. In reality, f will sometimes be off, and can lead the search astray. 迷途
- 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 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

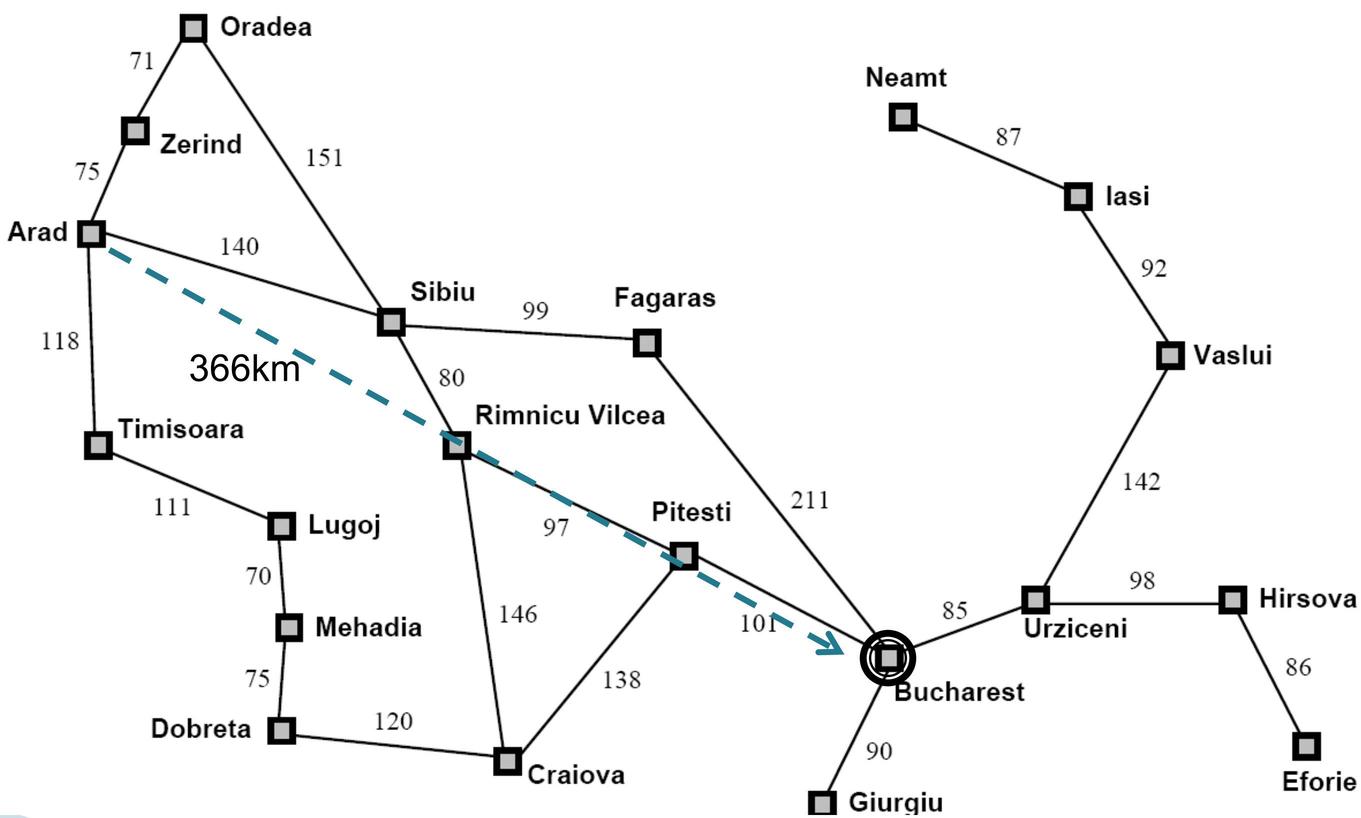
- Formally speaking, h can be any function at all, as long as $h(n) = 0$ if n is a goal.

- 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".

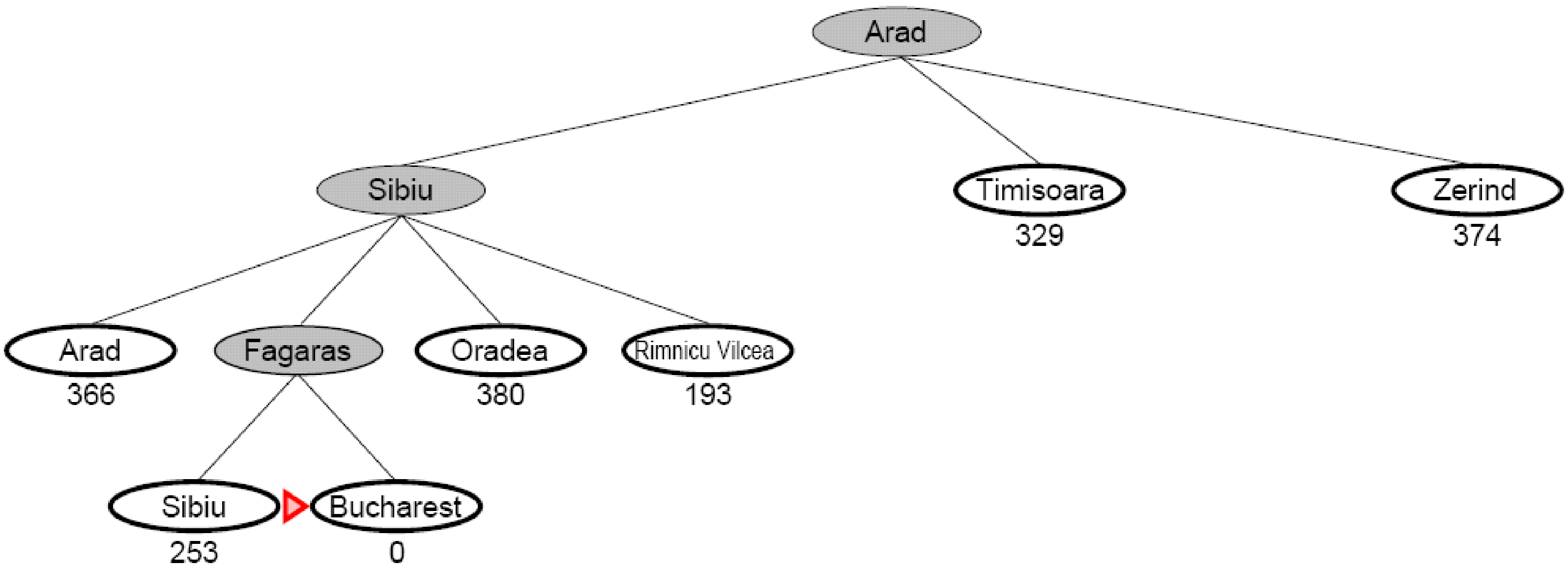
Romania with step costs in km



Straight−line distance
to Bucharest

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

Greedy search example



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

可採納的

A* search uses an admissible heuristic

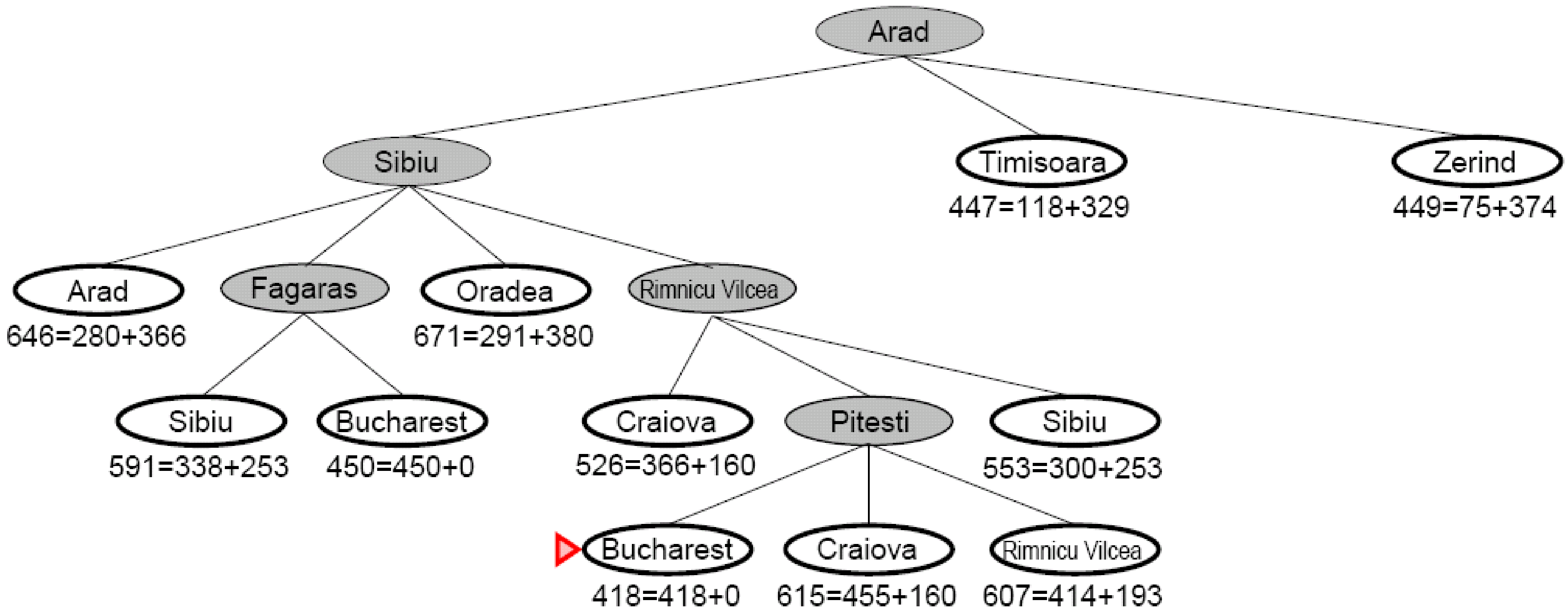
i.e. $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n

(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 <u>optimal path</u> 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 selects a suboptimal goal (G_2) for expansion

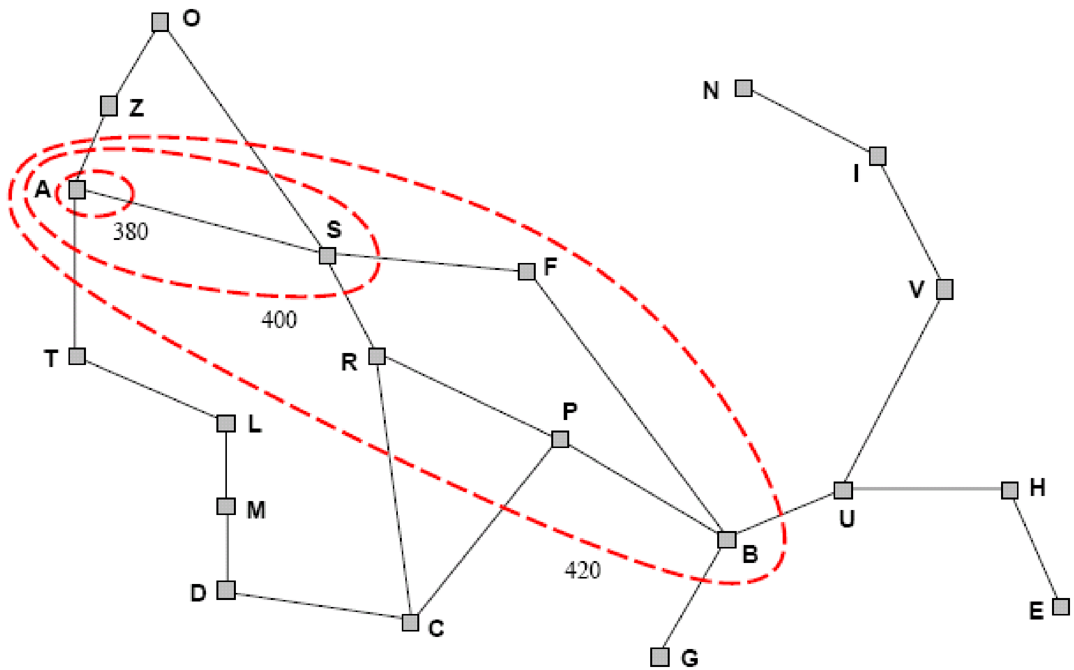
$\Rightarrow$ A* is optimal.

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

$$f_i \leq f(G)$$

Proof of lemma: Consistency

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

A heuristic is consistent if

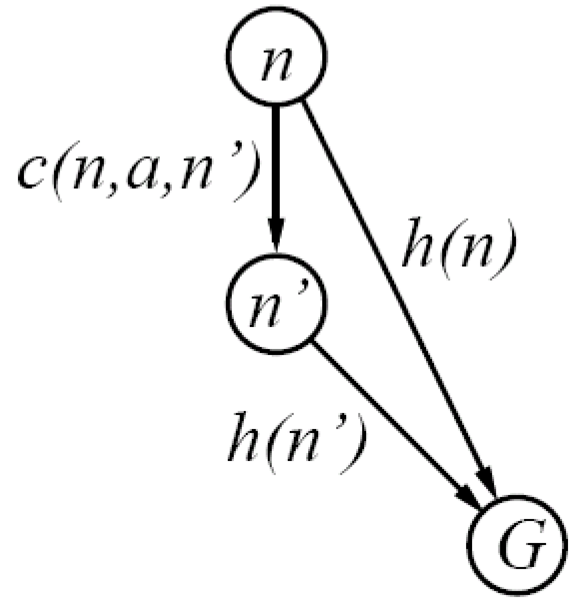
$h(n) \le c(n, a, n') + h(n')$

If h is consistent, we have

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

$f(n') \ge f(n)$

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



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 1st 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

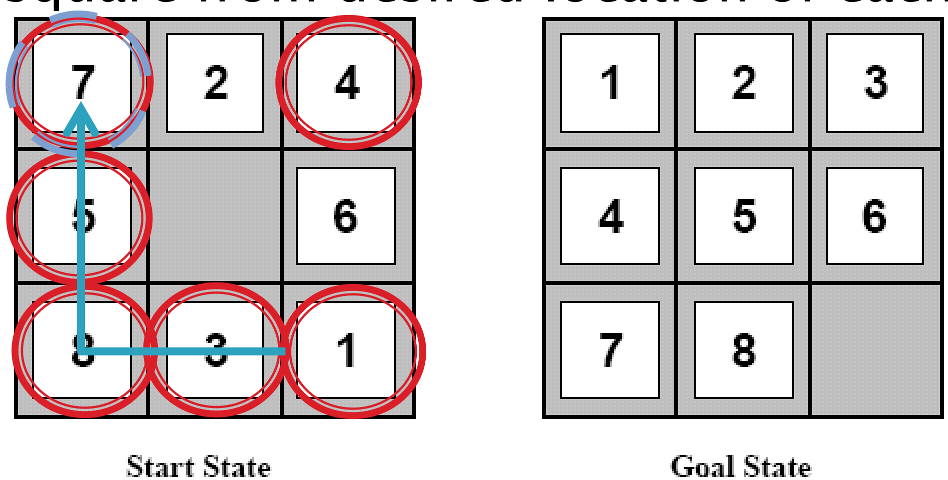
E.g. for the 8-puzzle

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance
(i.e. no. of square from desired location of each tile)

Misplaced tiles

'1' move to goal state need 4 steps



$h_1(S)$ =	6
$h_2(S)$ =	4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14

Dominance

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

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

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		
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.

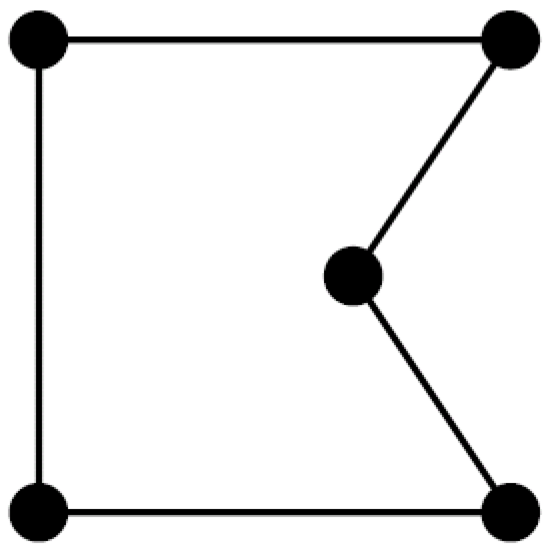
If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution.

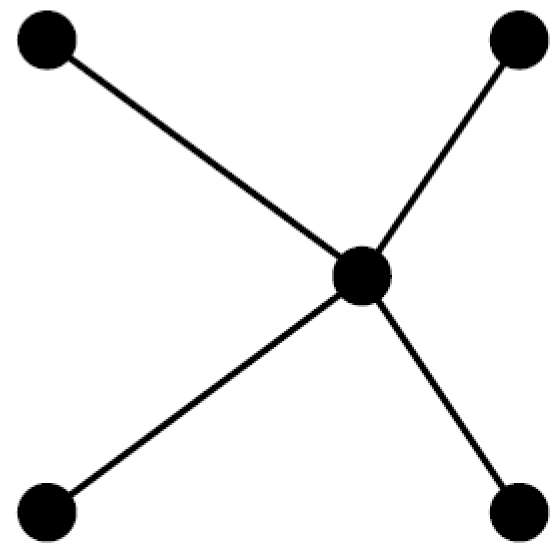
Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem admissible

Relaxed problems

Well-known example: travelling salesperson problem (TSP)

– Find the shortest tour visiting all cities exactly once





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

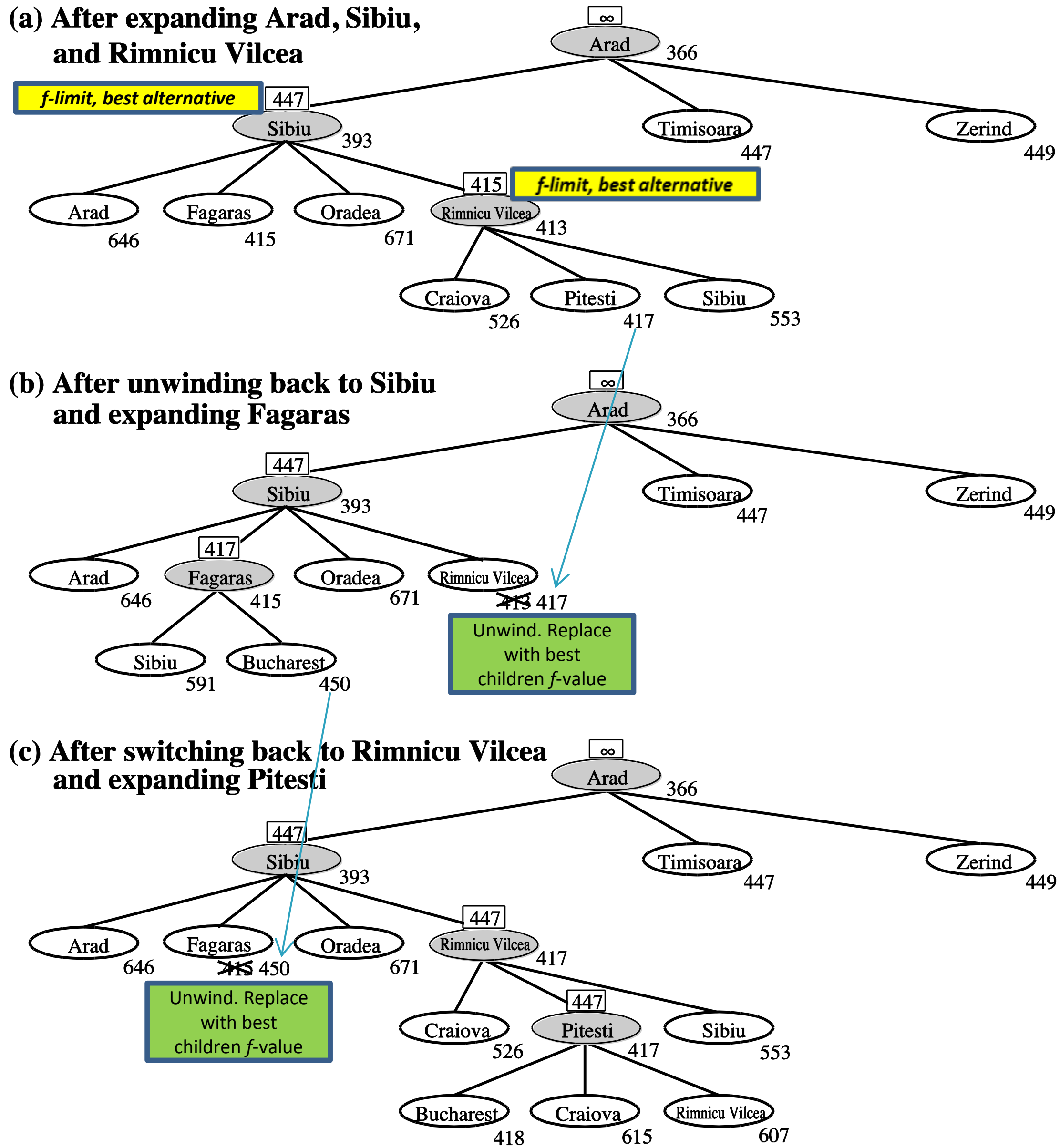
Memory-bounded Heuristic Search

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

- Iterative-deepening A* (**IDA***)
- Recursive best-first search (**RBFS**)
- Memory-bounded A* (**MA***)
- Simplified MA* (**SMA***)

The last 2 maximize the use of available memory

Recursive Best First Search (RBFS)

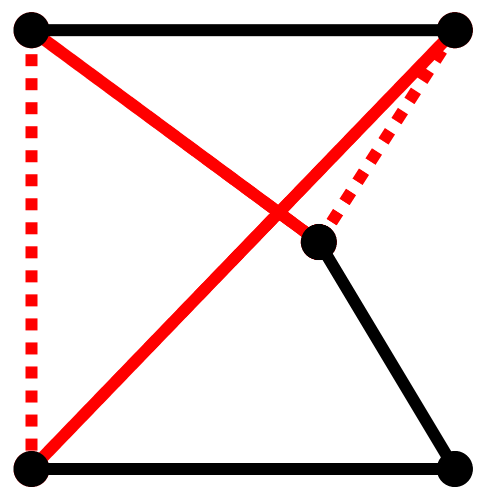


Iterative improvement algorithms

- 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 ??
- Then 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 as well as offline search

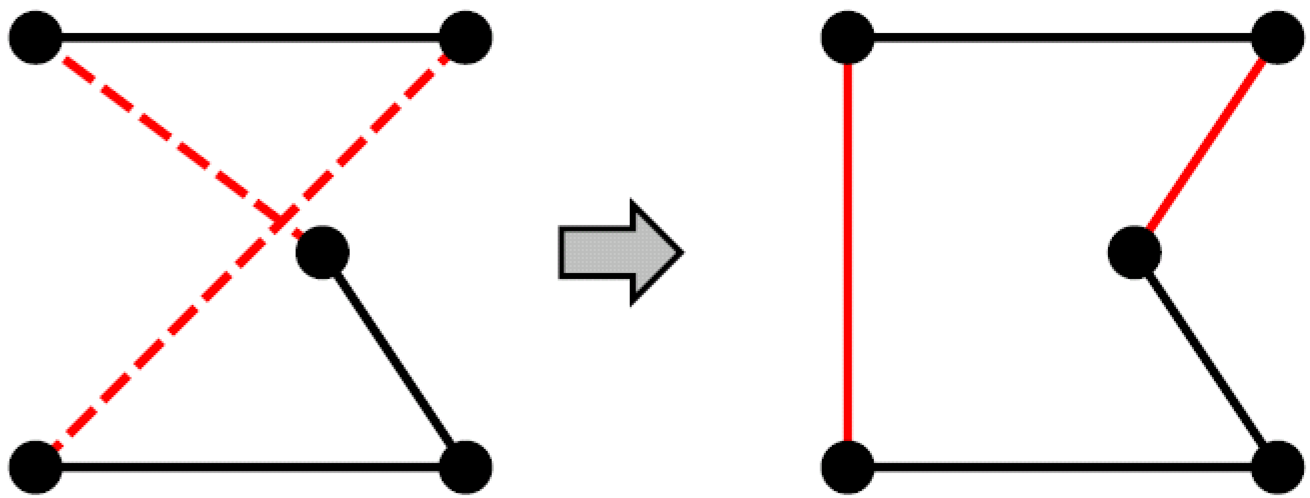
Example: Travelling Salesperson Problem

Start with any complete tour, perform pair-wise exchanges



Example: Travelling Salesperson Problem

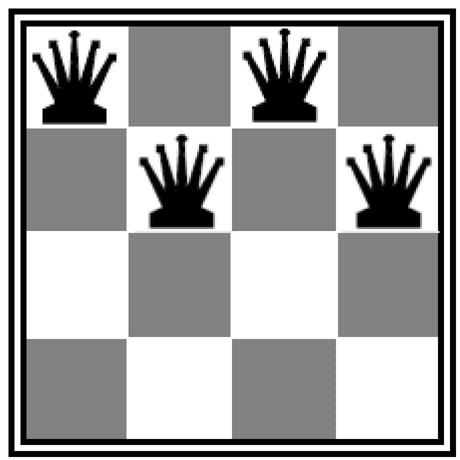
Start with any complete tour, perform pair-wise exchanges



Example: *n*-queens

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

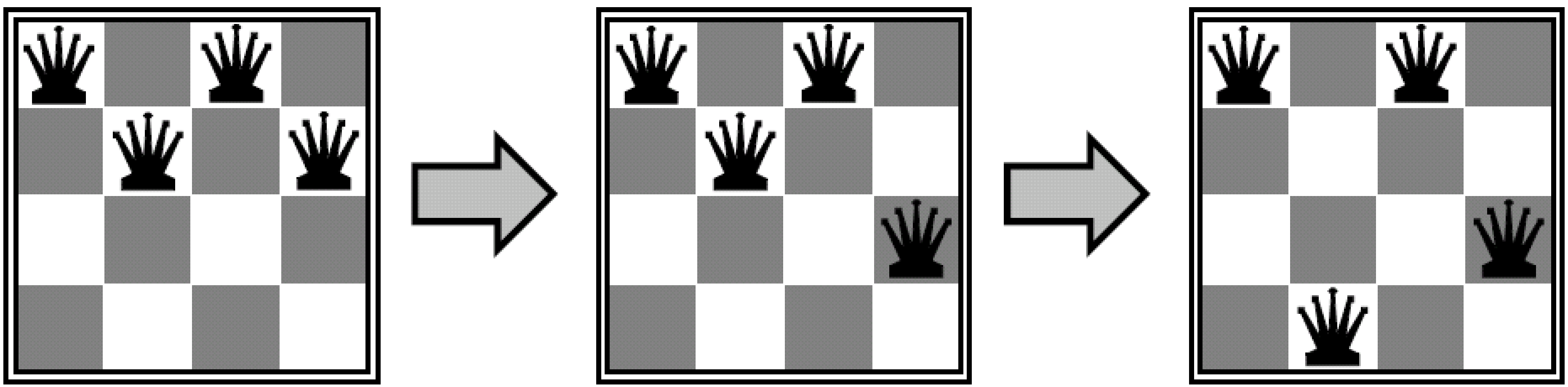
- Move a queen to reduce number of conflicts



Example: *n*-queens

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

- Move a queen to reduce number of conflicts



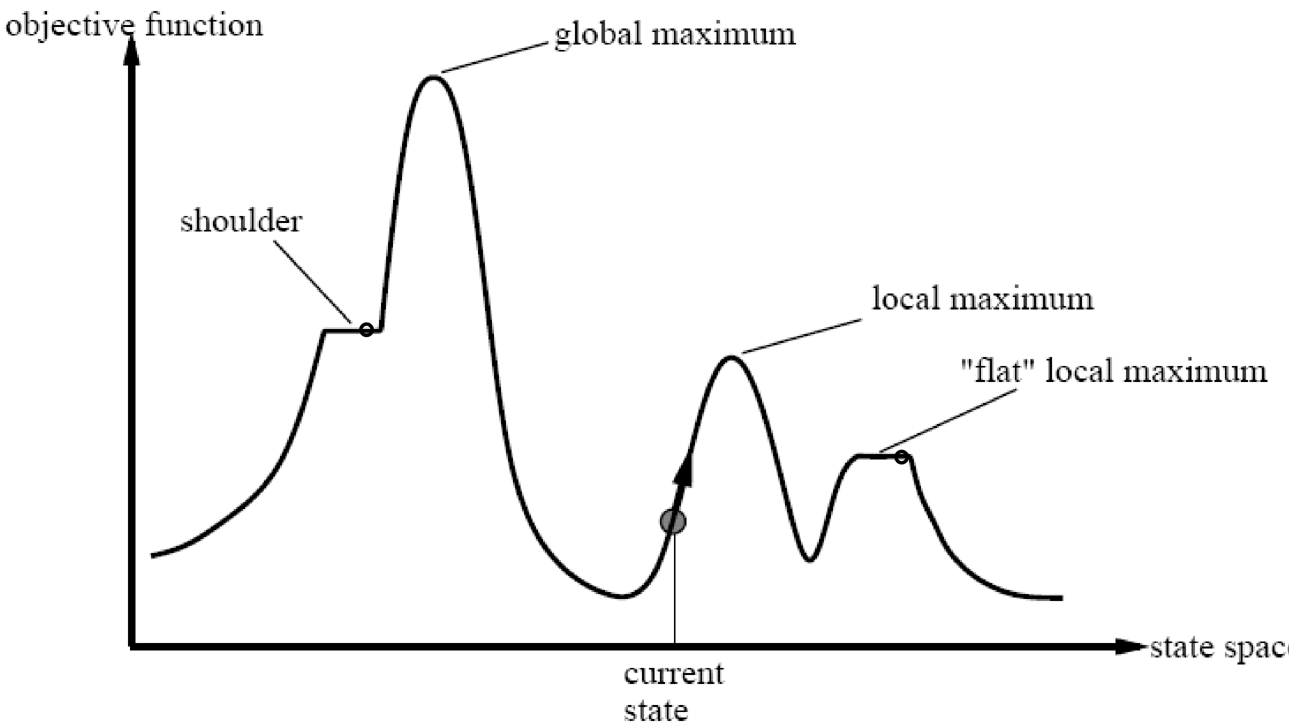
Hill Climbing (or gradient ascent / descent)

"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 ← Make-Node (Initial-State[*problem*])
  loop do
    neighbor ← a highest-valued successor of *current*
    if Value[*neighbor*] < Value[*current*] then **return** State[*current*]
    current ← *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? 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
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling probability of downward steps
   current ← Make-Node(Initial-State[problem])
   for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← Value[next] – Value[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

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 ΔE (-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

- The Value function (→ ΔE) 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.

At fixed "temperature" *T*, state occupation probability reaches Boltzmann distribution

$$p(x) = \alpha e^{E(x)/kT}$$

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?

Demo->

Genetic algorithms

- Genetic algorithm (GA): stochastic group search for global optimum (optima).
- Initial population: a set 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

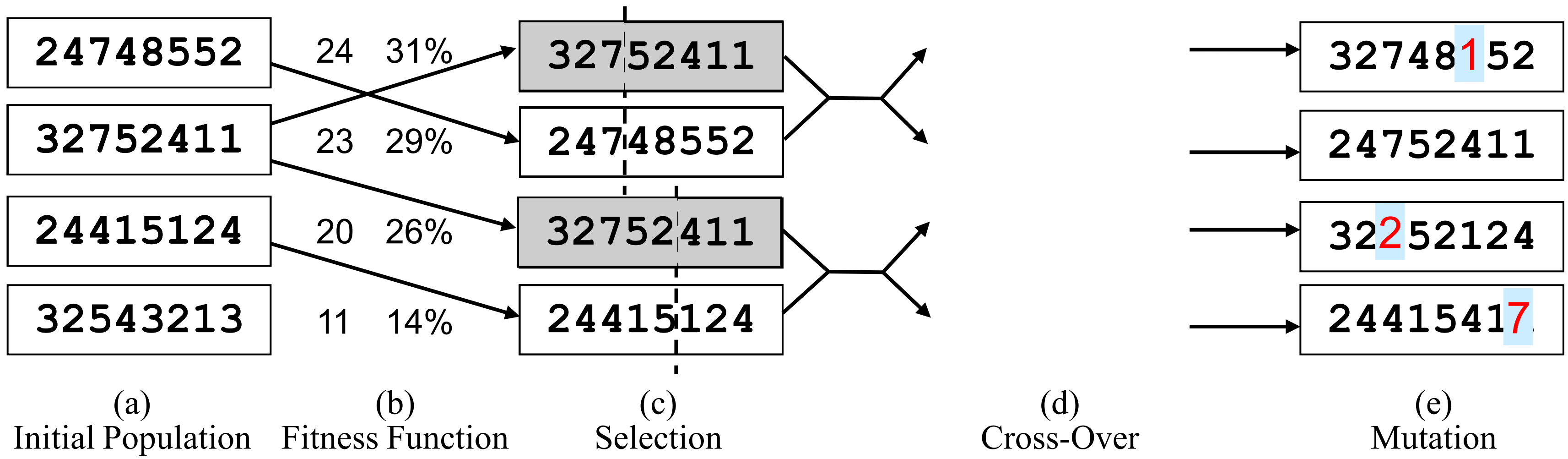


Fig 4.15 The GA. The initial population in (a) is ranked by the fitness function in (b). Resulting the pairs of mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Genetic Algorithm

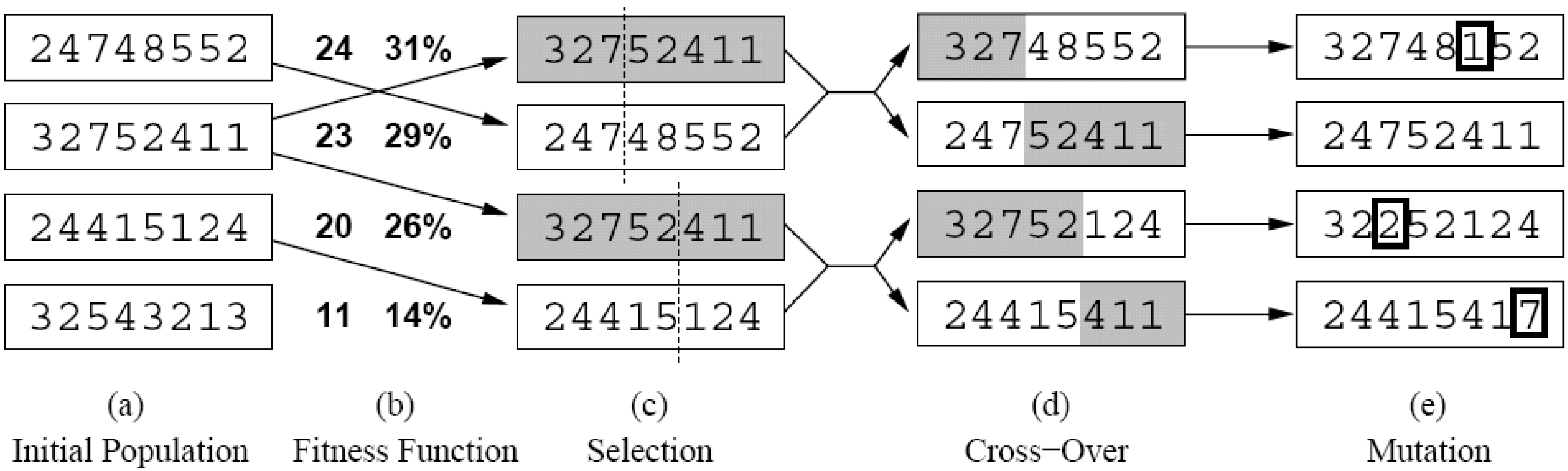
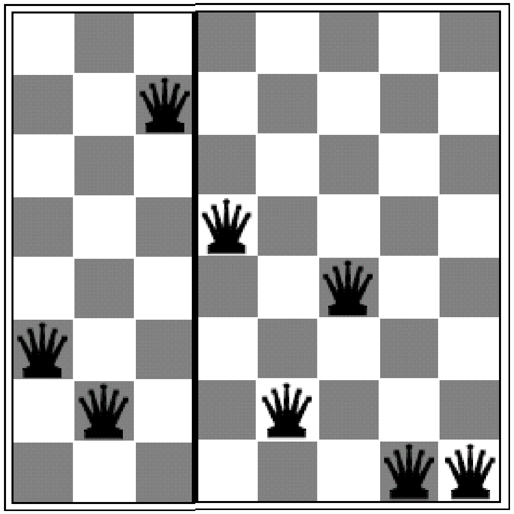


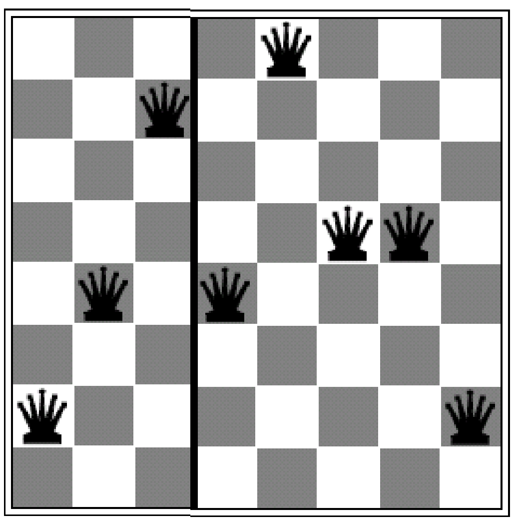
Fig 4.15 The GA. The initial population in (a) is ranked by the fitness function in (b). Resulting the pairs of mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Genetic Algorithm



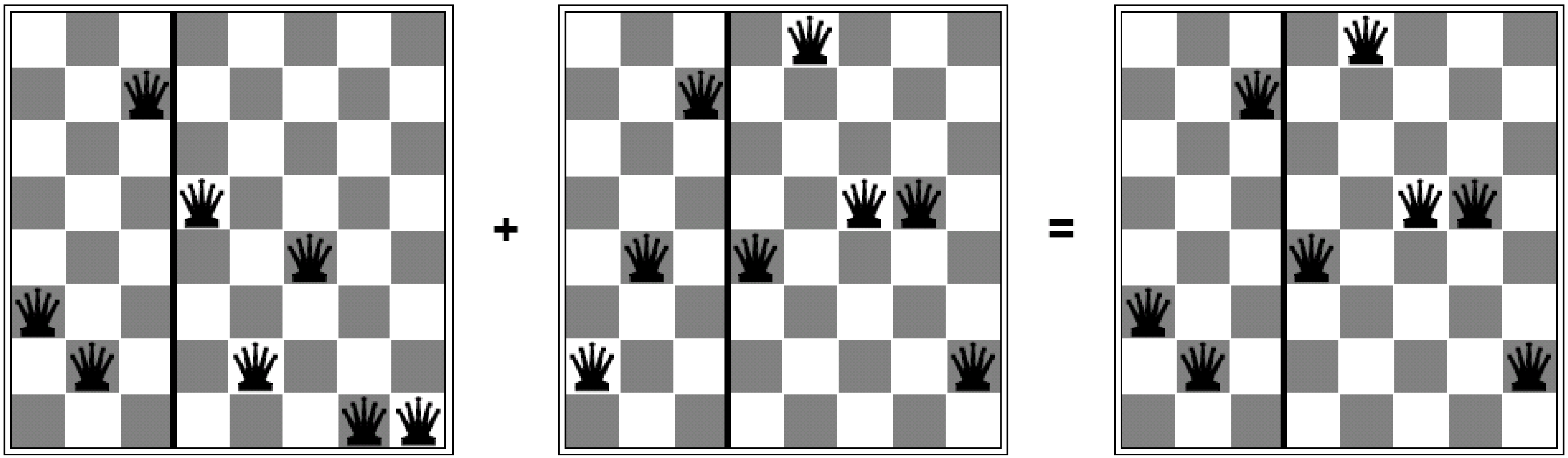
+

=



2 parents in Fig 4.15(c)
and 1st offspring in Fig 4.15 (d)

Genetic Algorithm



2 parents in Fig 4.15(c) and 1st 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 scatted 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.

<--Demo-->

```
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 ← empty set
      loop for i from 1 to Size(population) do
         x ← Random-Selection(population, Fitness-Fn) // fitness proportionate
         y ← Random-Selection(population, Fitness-Fn)
         child ← Reproduce(x,y) // crossover
         if (small random probability) then child ← Mutate(child) // mutation
         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 // crossover
   inputs: x, y, parent individuals

   n ← Length(x)
   c ← 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.

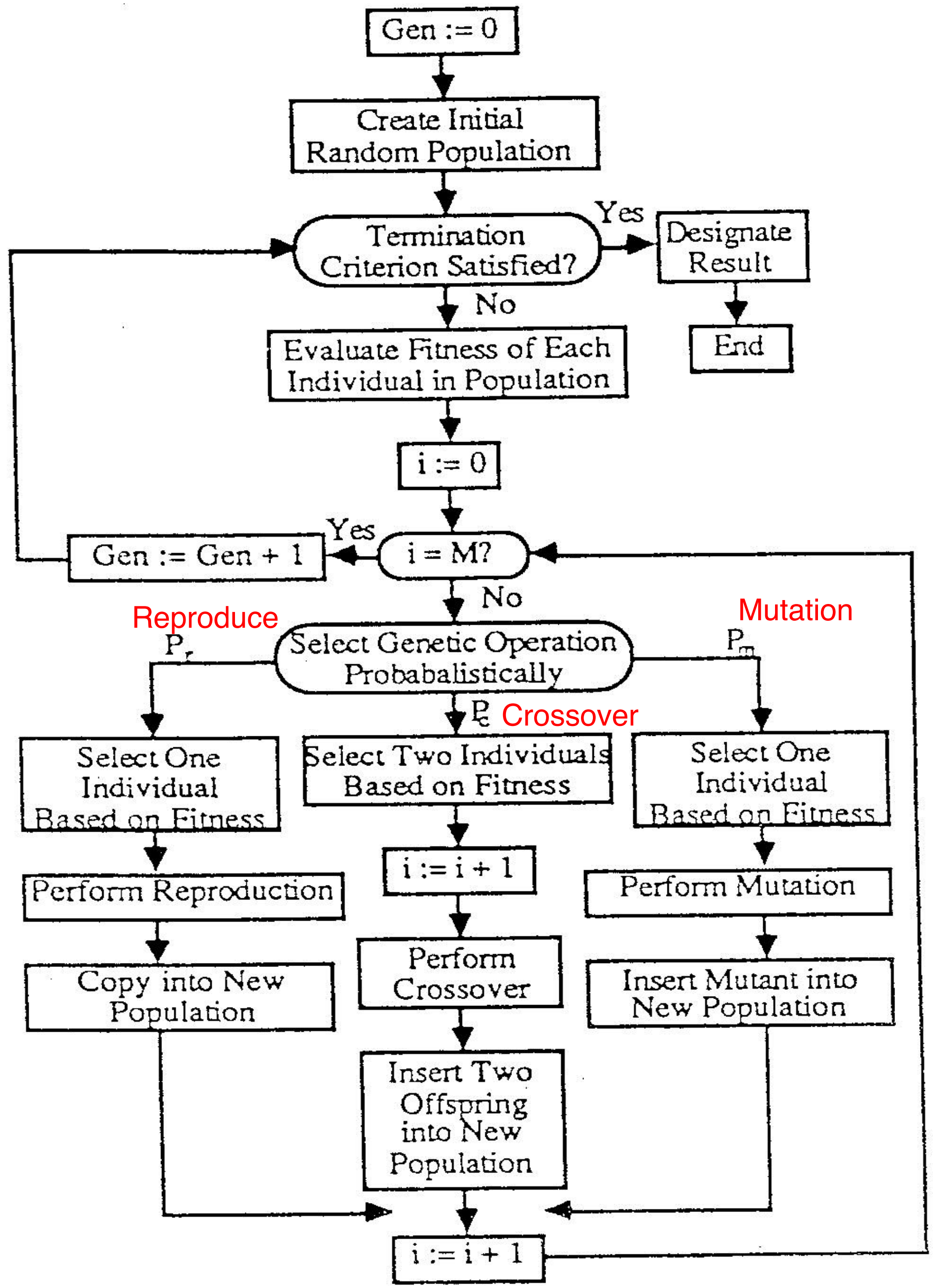


Figure 3.1 Flowchart of the conventional genetic algorithm.

M: population size

P: probability