02AB – ARTIFICIAL INTELLIGENCE

Lecture 3: Informed Search Algorithms

Soan Duong
soanduong@lqdtu.edu.vn
Computer Science Department, Le Quy Don Technical University

Outline

1. Introduction
2. Greedy best-first search
3. A* search
4. Heuristic functions
5. Hill-climbing search
6. Local beam search
7. Genetic algorithm

Introduction

- **Informed search** strategy: hint about the location of goals that came in the form of a heuristic function *h(n)*.

→ find solutions more efficiently than uninformed strategy!

- *h(n)*: estimated cost of the cheapest path from the state at node n to a goal state.

- E.g., estimate the distance from the current state to a goal by computing the straight-line distance on the map between the two points in route-finding problems.

Reminder best-first search alg.

- Which node from the frontier to expand next?
- → choose a node, *n*, with minimum values of some **evaluation function**, *f(n)*.

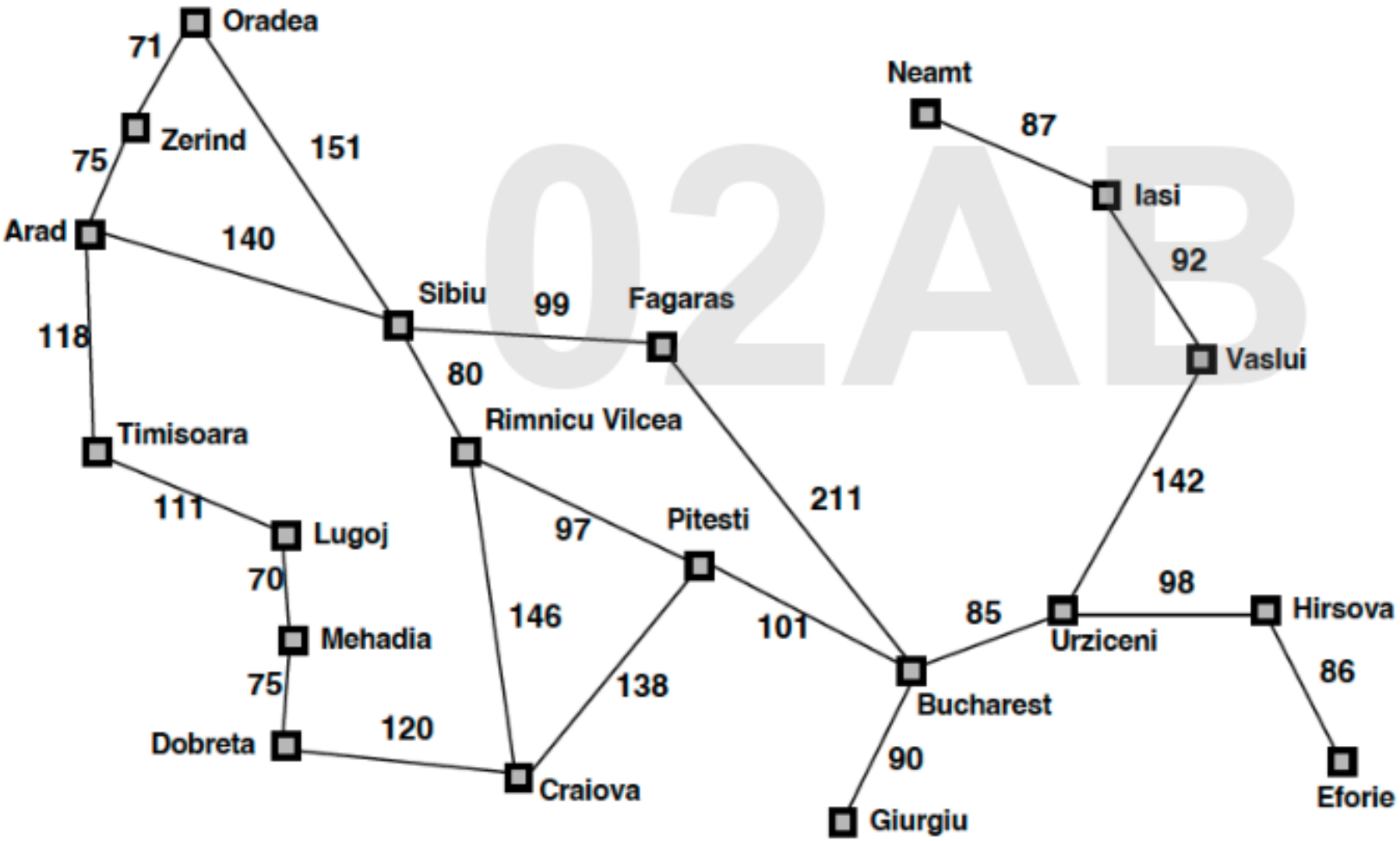
```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Greedy best-first search

- A form of `best-first search` that expands first the node that appears to be **closest** to the goal.
  - f(n) = h(n): distance to the goal
  - Choose a node with the lowest h(n)
  - "*Greedy*": tries to get as close to a goal as it can in each iteration
- In Romania problem, h(n): straight-line distance heuristics

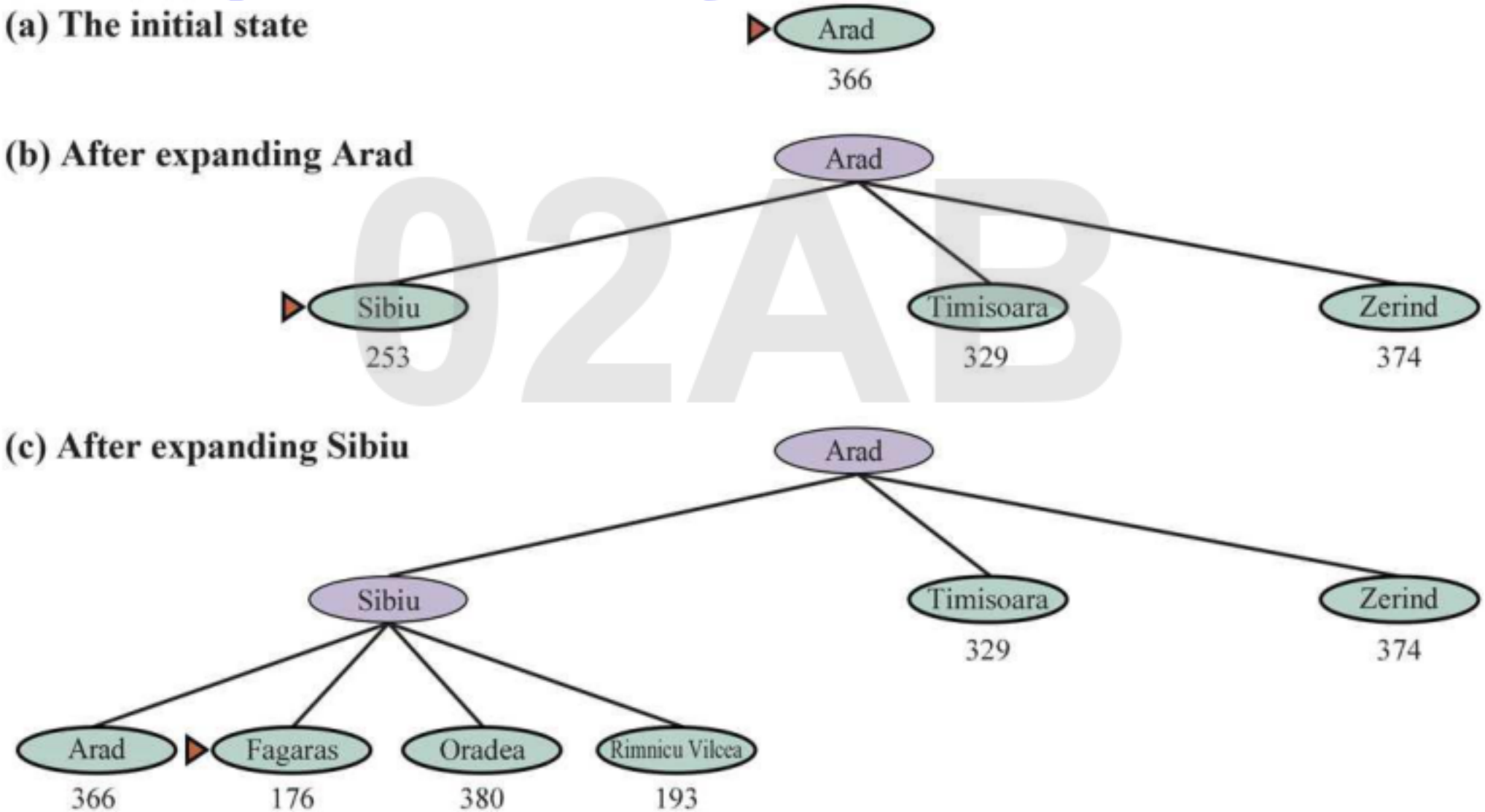
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Romania with step cost in km



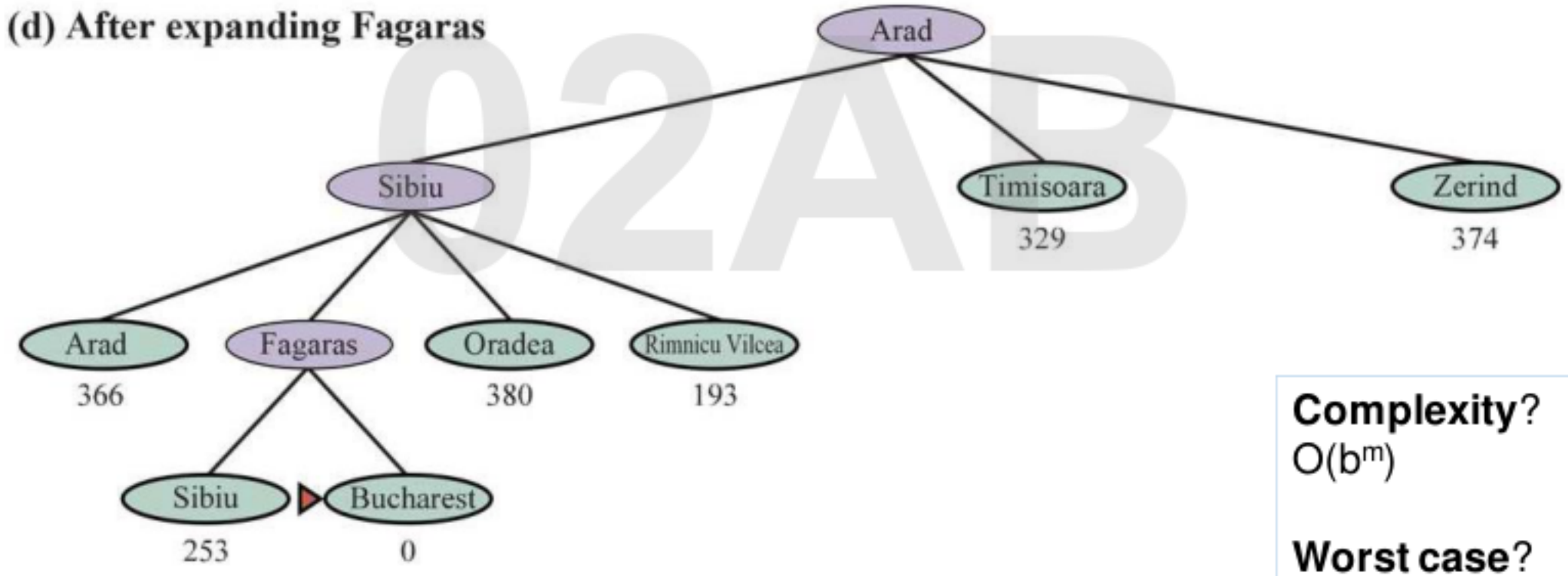
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: Greedy best-first search



Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic.

Example: Greedy best-first search

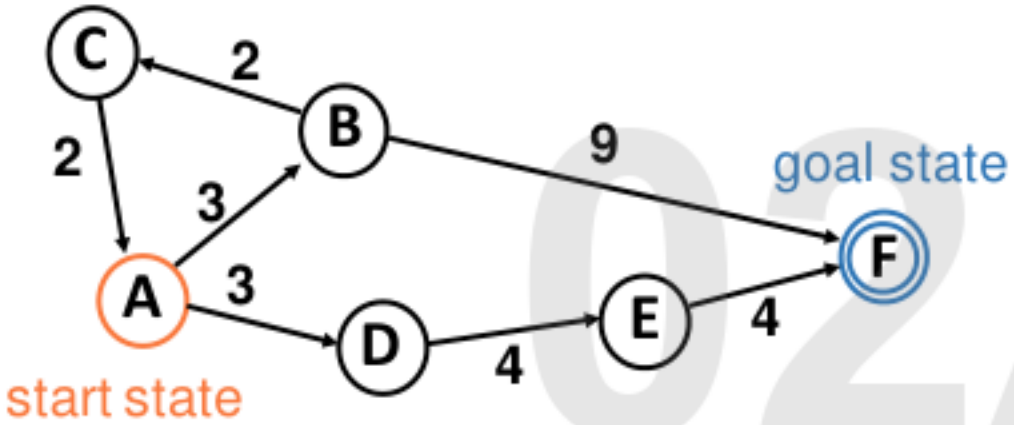


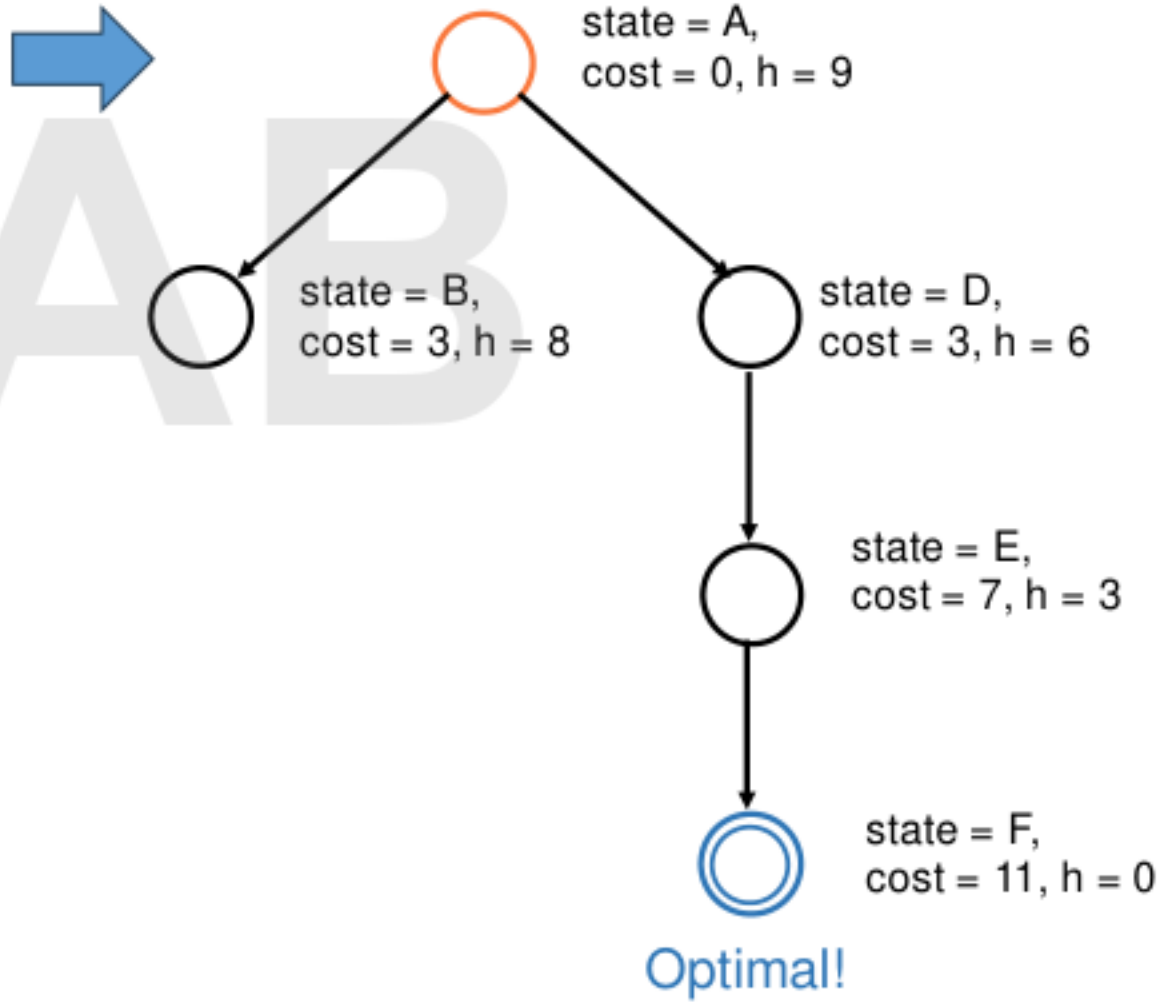
Complexity?
$O(b^m)$

Worst case?

The last stags in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic.

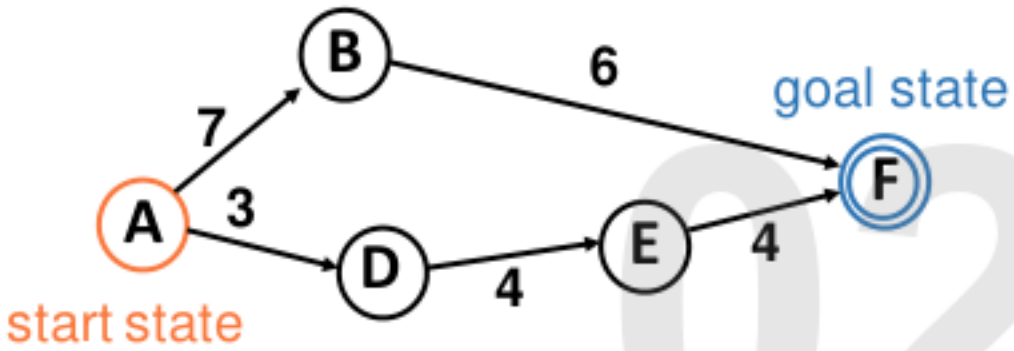
Example: Greedy best-first search

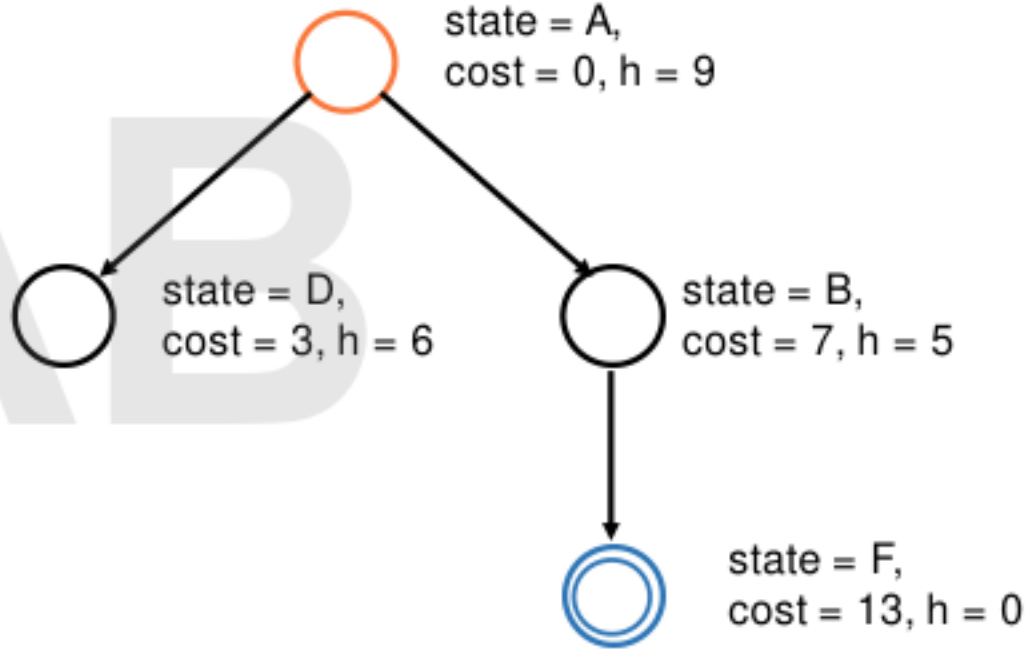




h(n)	Est. value
h(A)	9
h(B)	8
h(C)	9
h(D)	6
h(E)	3
h(F)	0

Example: Greedy best-first search





Non-optimal!

h(n)	Est. value
h(A)	9
h(B)	5
h(D)	6
h(E)	3
h(F)	0

Problem: greedy evaluates the promise of a node only by how far is left to go, and does not take the cost that occurred already into account!

Properties of greedy search

- **Complete**: No, can get stuck in loops

→Complete in finite space with repeated-state checking.

- **Time**: $O(b^m)$

A good heuristic can give dramatic improvement.

- **Space**: $O(b^m)$, keep all nodes in memory.

- **Optimal**: No

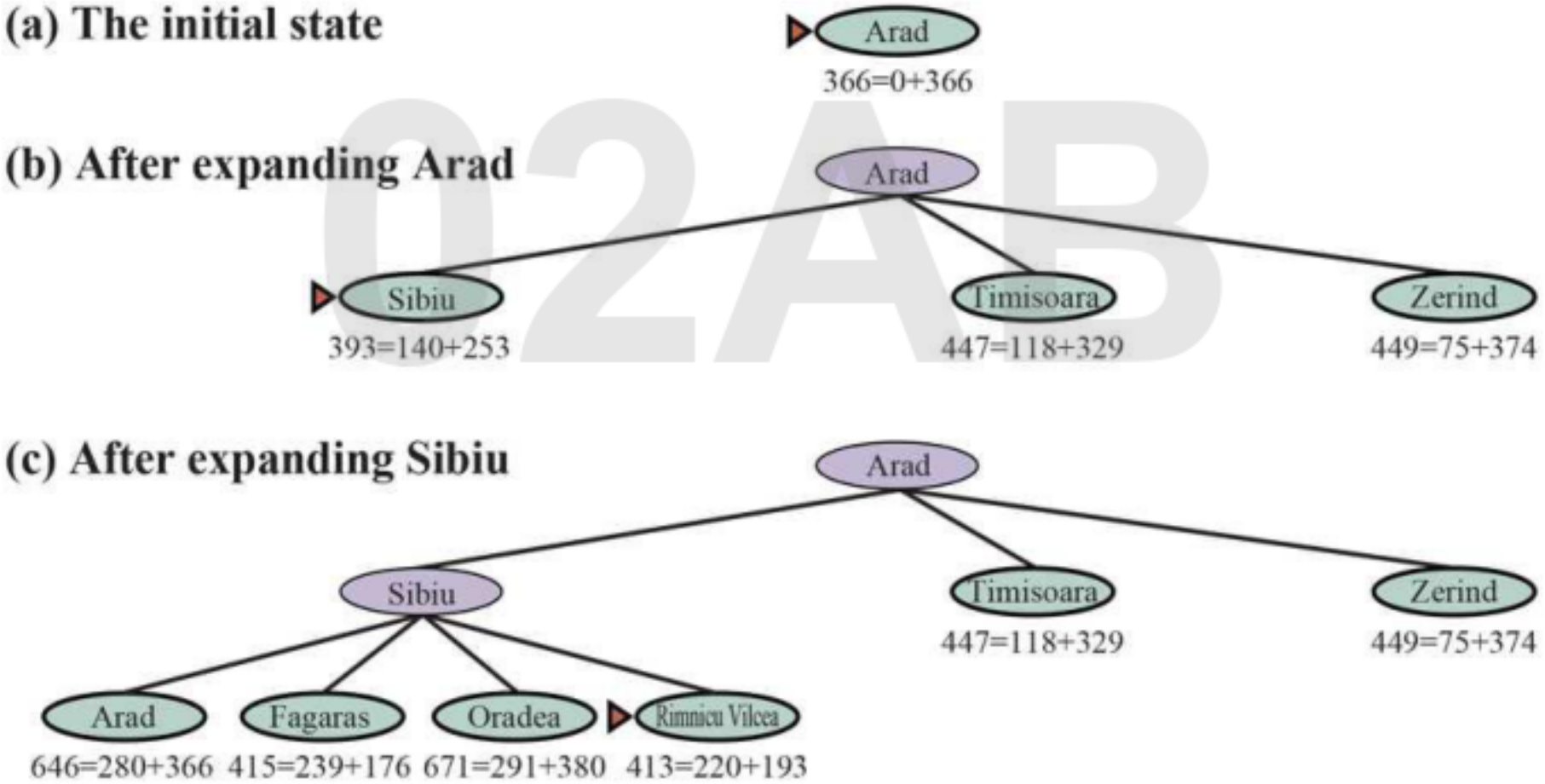
A* search

- The most common informed search algorithm.
- Idea: avoid expanding paths that are already expensive.
- A form of `best-first search` that uses:

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

  - *g(n)*: the path cost from the initial state to node *n.*
  - *h(n)*: the estimated cost to the goal from *n*.
  - →*f(n)*: estimated cost of the path through *n* to the goal.
- Use an **admissible heuristic**, i.e., $h(n) \leq h^*(n)$
  - $h^*(n)$: true cost from n to the goal.

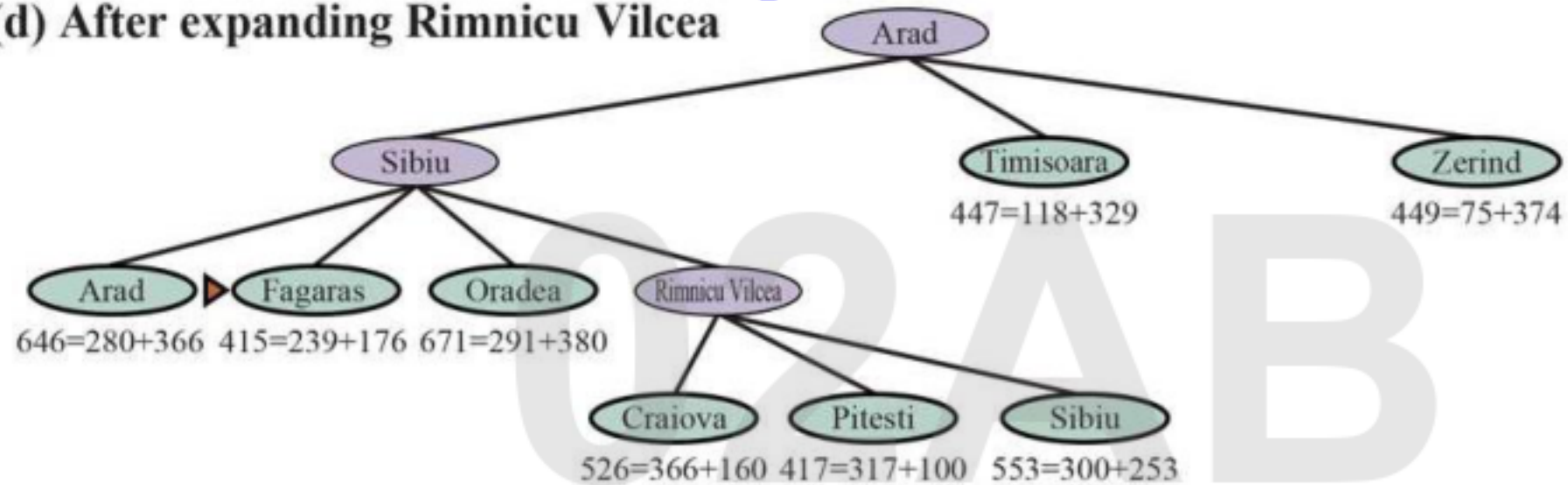
Example: A* search



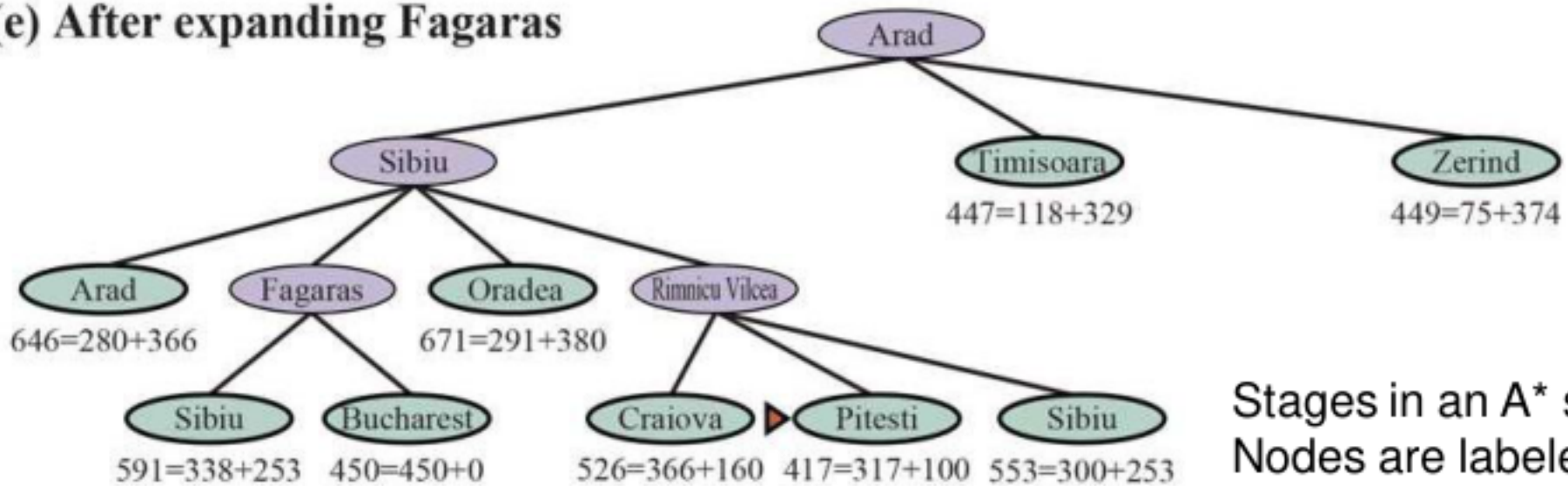
Stages in an A* search for Bucharest. Nodes are labeled with f = g + h.

Example: A* search

(d) After expanding Rimnicu Vilcea

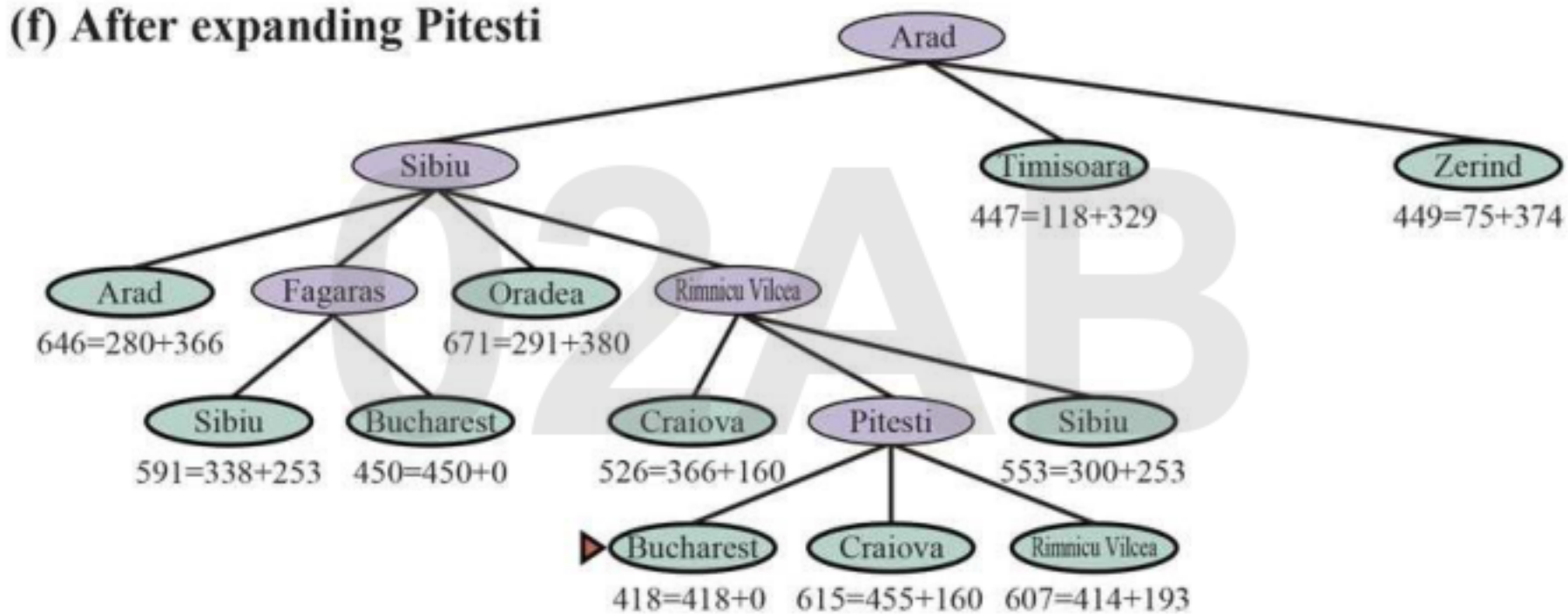


(e) After expanding Fagaras



Stages in an A* search for Bucharest. Nodes are labeled with f = g + h.

Example: A* search



The last stages in an A* search for Bucharest. Nodes are labeled with f = g + h.

- A* returns the cost-optimal path!

Optimality of A* search

- If the heuristic is admissible, A* is optimal.
- Proof by contradiction:
  - Suppose the optimal path has cost C*.
  - The algorithm returns a suboptimal path with cost C > C*.
  - → there must be some node on the optimal path that is unexpanded.
  - g*(n): the cost of the optimal path from the start to n.
  - h*(n): the cost of the optimal path from to the nearest goal.
  - →

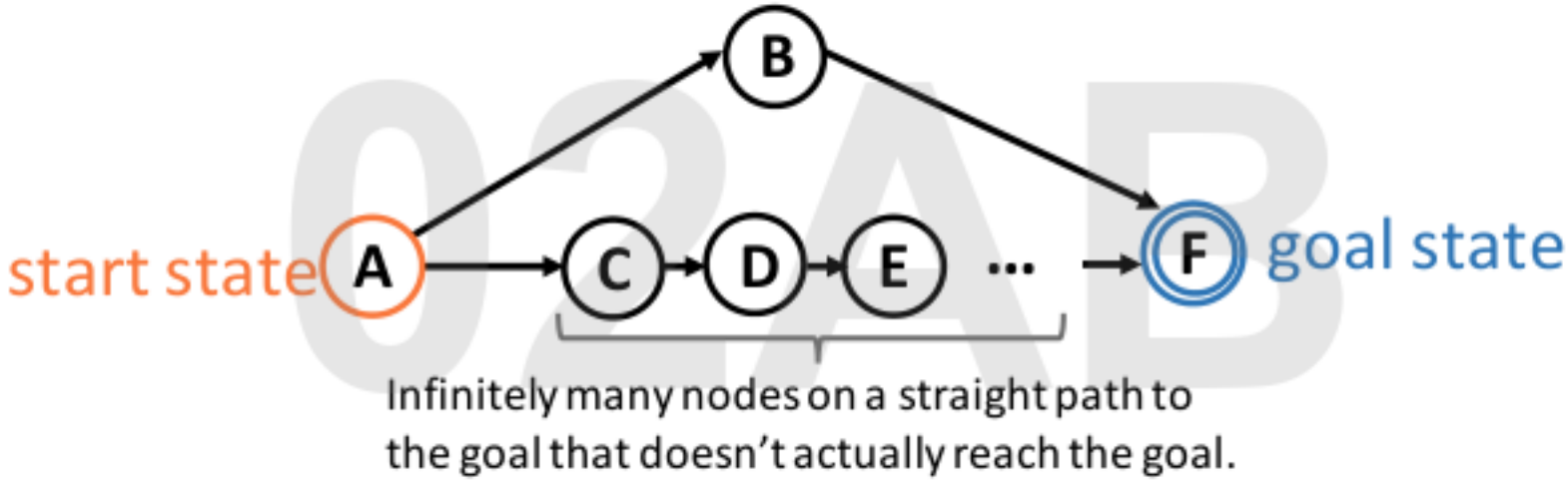
$$
\begin{aligned}
f(n) &> C^* \quad (\text{otherwise } n \text{ would have been expanded}) \\
f(n) &= g(n) + h(n) \quad (\text{by definition}) \\
f(n) &= g^*(n) + h(n) \quad (\text{because } n \text{ is on an optimal path}) \\
f(n) &\leq g^*(n) + h^*(n) \quad (\text{because of admissibility}, h(n) \leq h^*(n)) \\
f(n) &\leq C^* \quad (\text{by definition}, C^* = g^*(n) + h^*(n))
\end{aligned}
$$

Form a contradiction!

A* return only cost-optimal paths.

Properties of A* search

- **Complete**: Yes, unless there are infinitely many nodes.



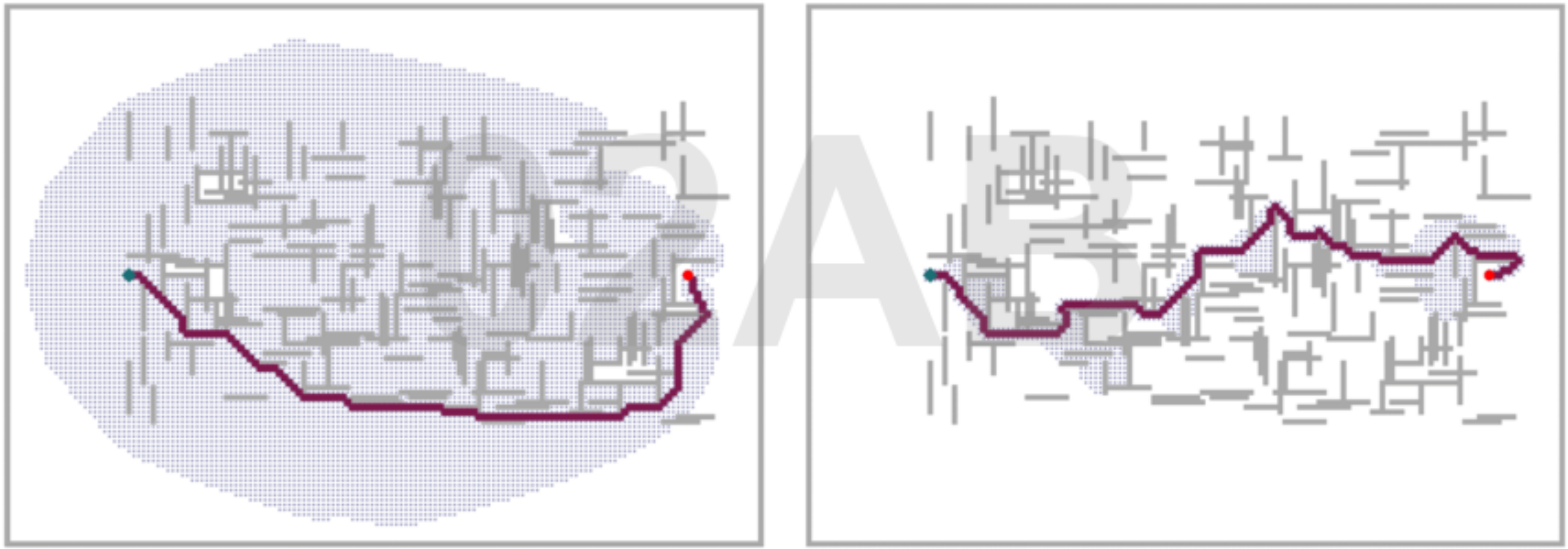
- **Time**: $O(b^m)$.
- **Space**: $O(b^m)$, keep all nodes in memory.
- **Optimal**: Yes

Weighted A* search

- A* search expands a lot of nodes.
- Satisficing solutions: explore fewer nodes with accept suboptimal solutions.
- **Detour index** concept of road engineers: multiplier applied to the straight-line distance to account for the typical curvature of roads.
- → Weighted A* search:

$$f(n) = g(n) + W \times h(n), \text{ for some } W > 1.$$

A* vs. weighted A* searches



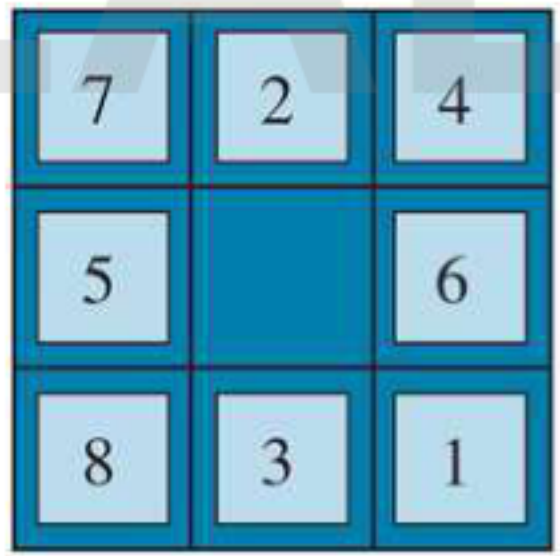
(a) (b)

Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight W=2 The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

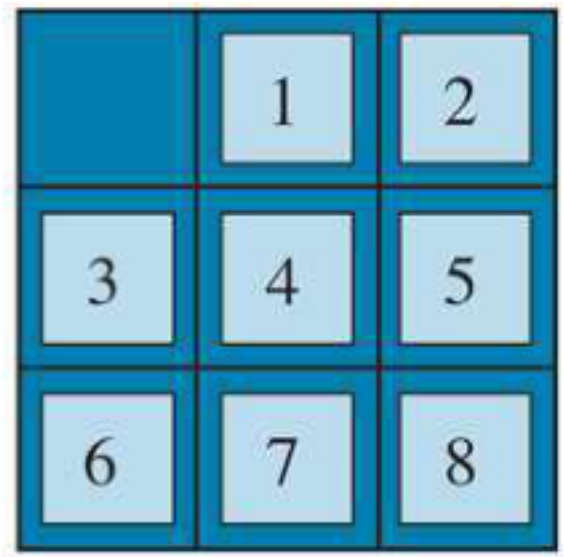
Heuristic functions

- N-puzzle problems
  - h1: the number of misplaced tiles (blank not included).
  - h2: the sum of the distances of the tiles from their goal positions (Manhattan distance = number of the moves to the goal).

- h1 = ?
- h2 = ?
- M(1) = M((3,3), (1,2)) = 3
- M(2) = M((1,2), (1,2)) = 1
- h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18



Start State



Goal State

Designing heuristic

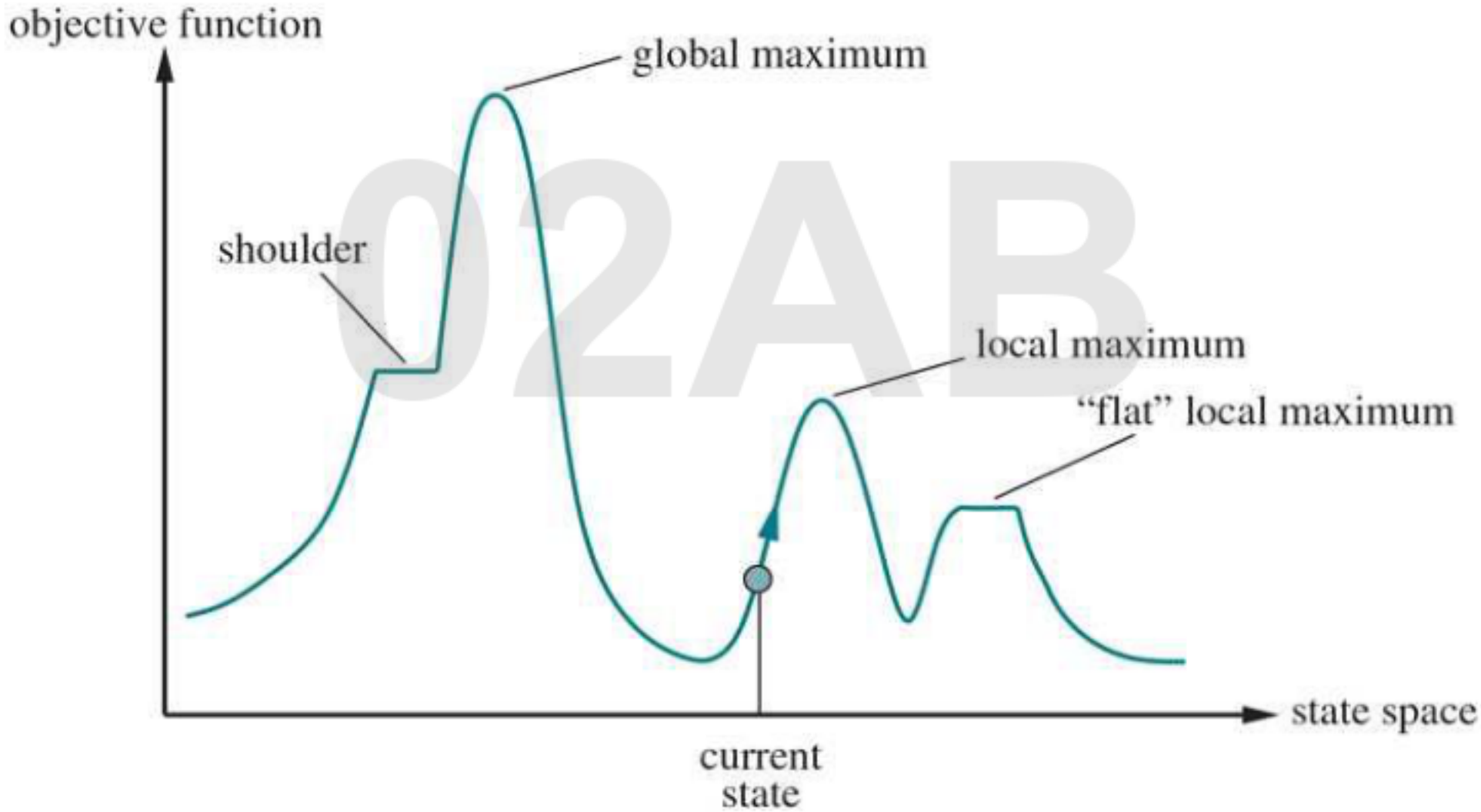
- One strategy for designing heuristics: relax the problem (make it easier)
- *"Number of misplaced tiles"* heuristic corresponds to a relaxed problem where tiles can jump to any location, even if something else is already there
- *"Sum of Manhattan distances"* corresponds to a relaxed problem where multiple tiles can occupy the same spot
- Another relaxed problem: only move 1,2,3,4 into the correct locations
- The ideal relaxed problem is
  - easy to solve,
  - not much cheaper to solve than the original problem.
- Some programs can successfully automatically create heuristics.

Search in complex environments

Local search algorithms

- Operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached.
- → might never explore a portion of the search space where a solution actually resides.
- Advantages:
  - Use very little memory; and
  - Can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

Example: 1D state-space landscape



A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

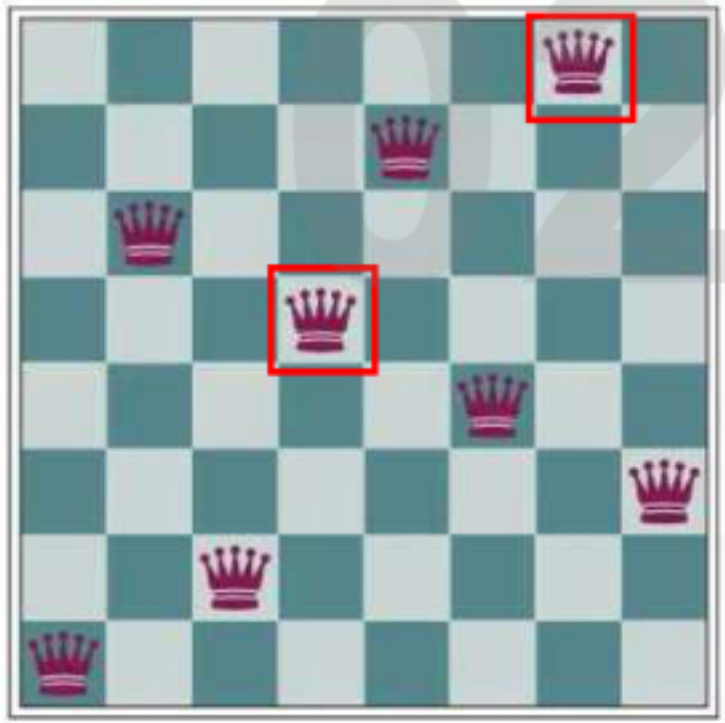
Hill-climbing search

- Known as gradient ascent/descent.
- Keeps track of one current state.
- On each iteration, moves to the neighboring state with the highest value, providing the steepest ascent.

```
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
```

Example: hill-climbing search

- ***The 8-queens problem**: place 8 queens on a chess board so that no queen attacks another.*



The heuristic cost function is the number of pairs of queens that are attacking each other. 8 moves are tied for best, with h=12. *The hill-climbing algorithm will pick one of these.*

Local beam search

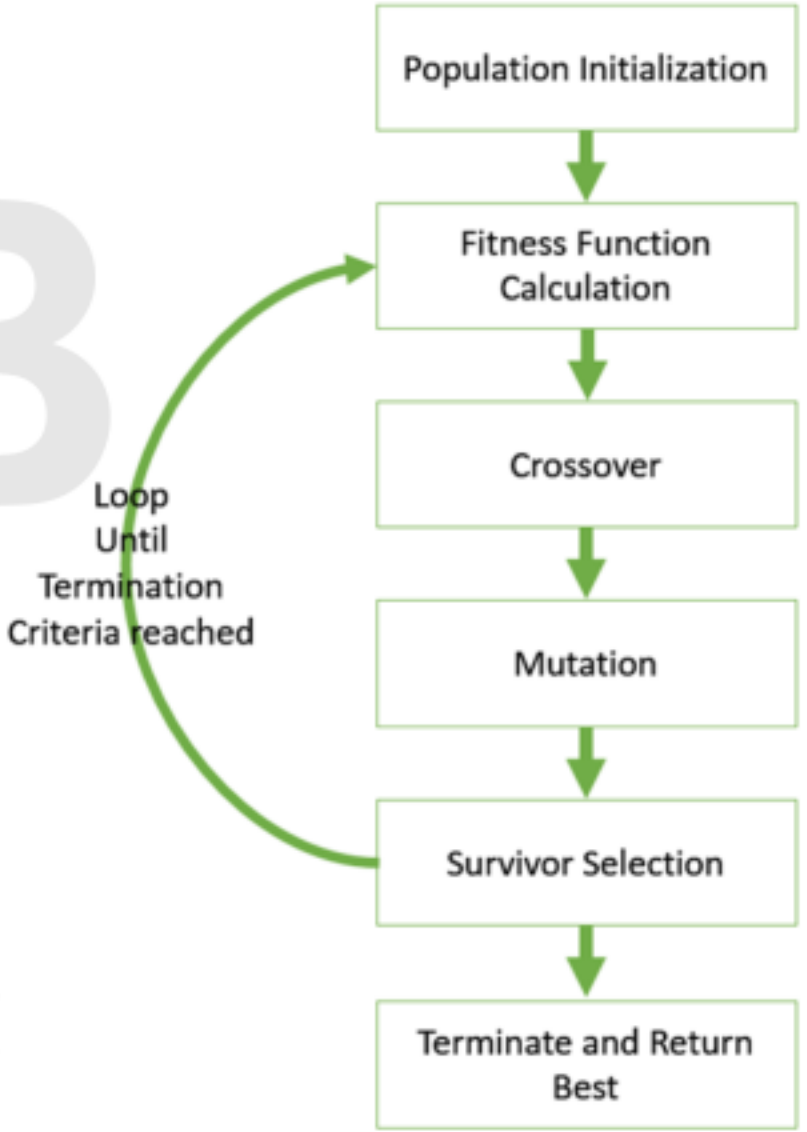
- Keeping just **one node** in memory might seem to be an extreme reaction to the problem of memory limitations.
- The **local beam search** algorithm keeps track of **k states** rather than just one.
- Algorithm:
  - Generate k random states
  - Generate the successors of all k states at each step.
  - If any successor is a goal, halt the algorithm.
  - Otherwise, select the k best successors from the complete list and repeat.

Genetic algorithms (GAs)

- A search technique used in computing to find **true** or **approximate** solutions to optimization and search problems. Known as **global** search heuristics.

- A particular class of evolutionary algorithms that are inspired by the metaphor of natural selection in biology:

  *There is a population of **individuals** (states), in which the **fittest** (highest value) individuals produce **offspring** (successor states) that populate the next generation, a process called **recombination**.*

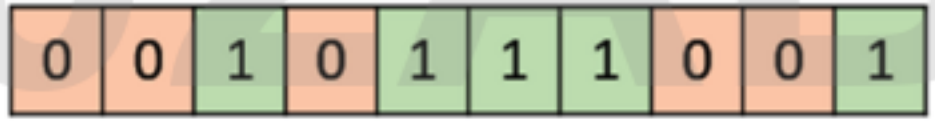


GA terminology

- **Individual**: any possible solution.
- **Population**: a group of all individuals.
- **Fitness function**: target function to optimize (each individual has a fitness).
- **Trait**: possible aspect (features) of an individual.
- **Genome**: collection of all chromosomes (traits) for an individual.
- **Encoding and decoding**: encode/decode the actual solution space to the computation (genome) space.
- **Genetic operators**: crossover, mutation, selection to composite the offspring.

Genotype representation

- **Binary representation**: consists of bit string
  - *Knapsack problem*: Given a set of items, each with a weight and a value, which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible?
  - 0: not picked; 1: picked



- **Integer representation**: suitable for problems where the solution can be expressed as a sequence of integers.
- **Permutation representation**, e.g. n-queens and TSP.
- **Real valued representation**, e.g. optimization of mathematical functions.

Population

- **Population** (set of **chromosomes**) is a subset of solutions in the current **generation**.
- Population **diversity**: should be maintained, otherwise leads to premature convergence.
- Population size:
  - Large: slow down,
  - Small: not enough for a good mating pool.
- Population initialization:
  - Random initialization: Populate with completely random solutions.
  - Heuristic initialization: Populate using a known heuristic for the problem.
- Population data structure: 2D array

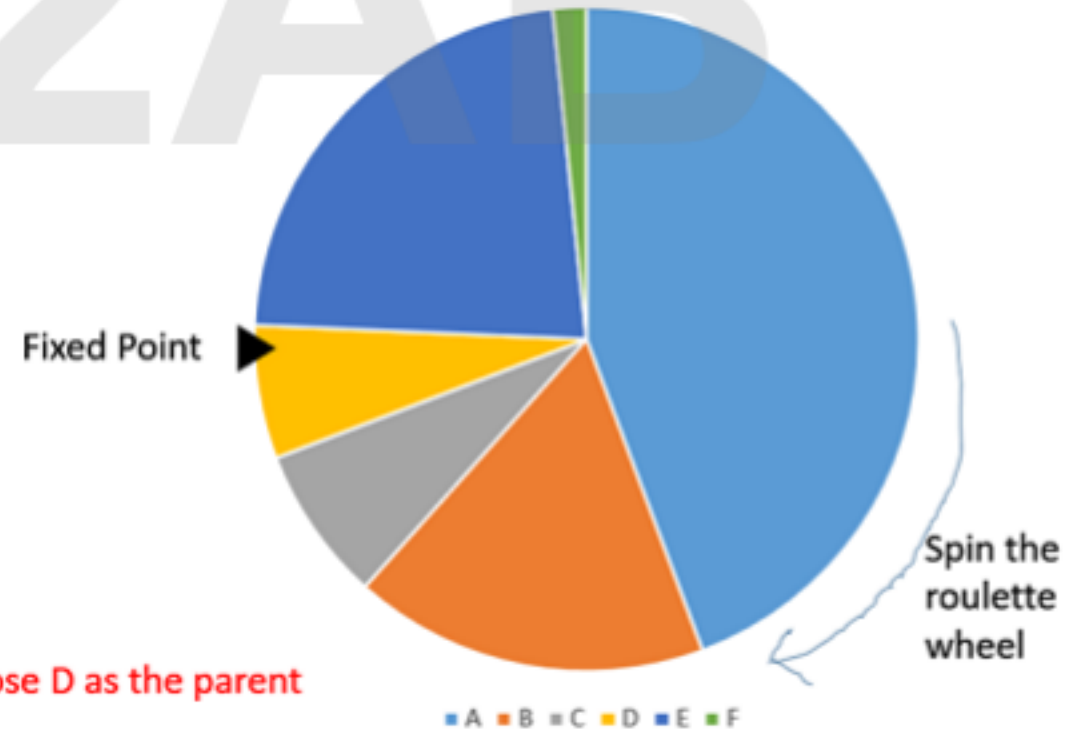
  population_size × chromosome size

Selection

- **Parent selection**: select parents which mate and recombine to create offsprings for the next generation.
  - *Fitness proportionate selection*: every individual can become a parent with a probability that is proportional to their fitness. E.g., Roulette wheel selection and stochastic universal sampling.
  - *Tournament selection*: select the best individual of a random subset.
  - *Rank selection*: select based on the individual's ranks.
  - *Random selection*: randomly select parents from the existing population.
- **Survivor selection**: determines which individuals are to be kicked out and kept in the next generation.
  - Age-based selection
  - Fitness-based selection

Roulette wheel selection

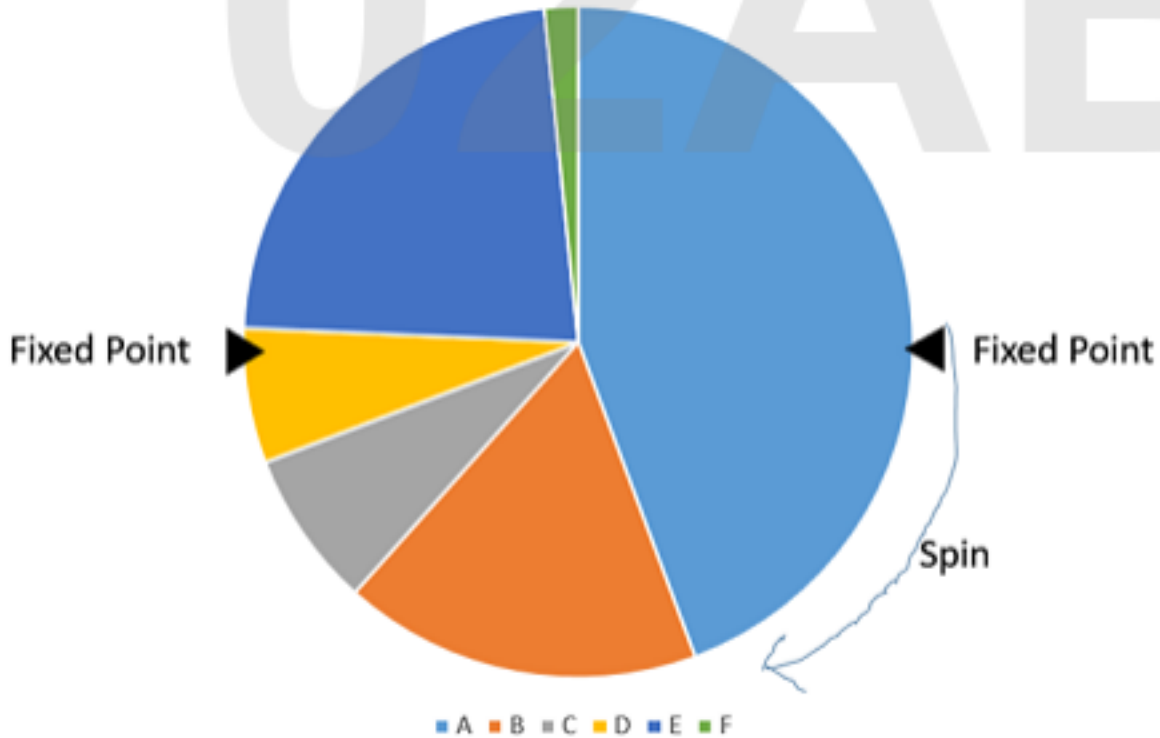
- The wheel is divided into **n pies**, where n is the number of individuals in the population.
- Set a fixed point.
- Spin the wheel to select one parent.
- Spin the wheel again to select another parent.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Stochastic universal sampling

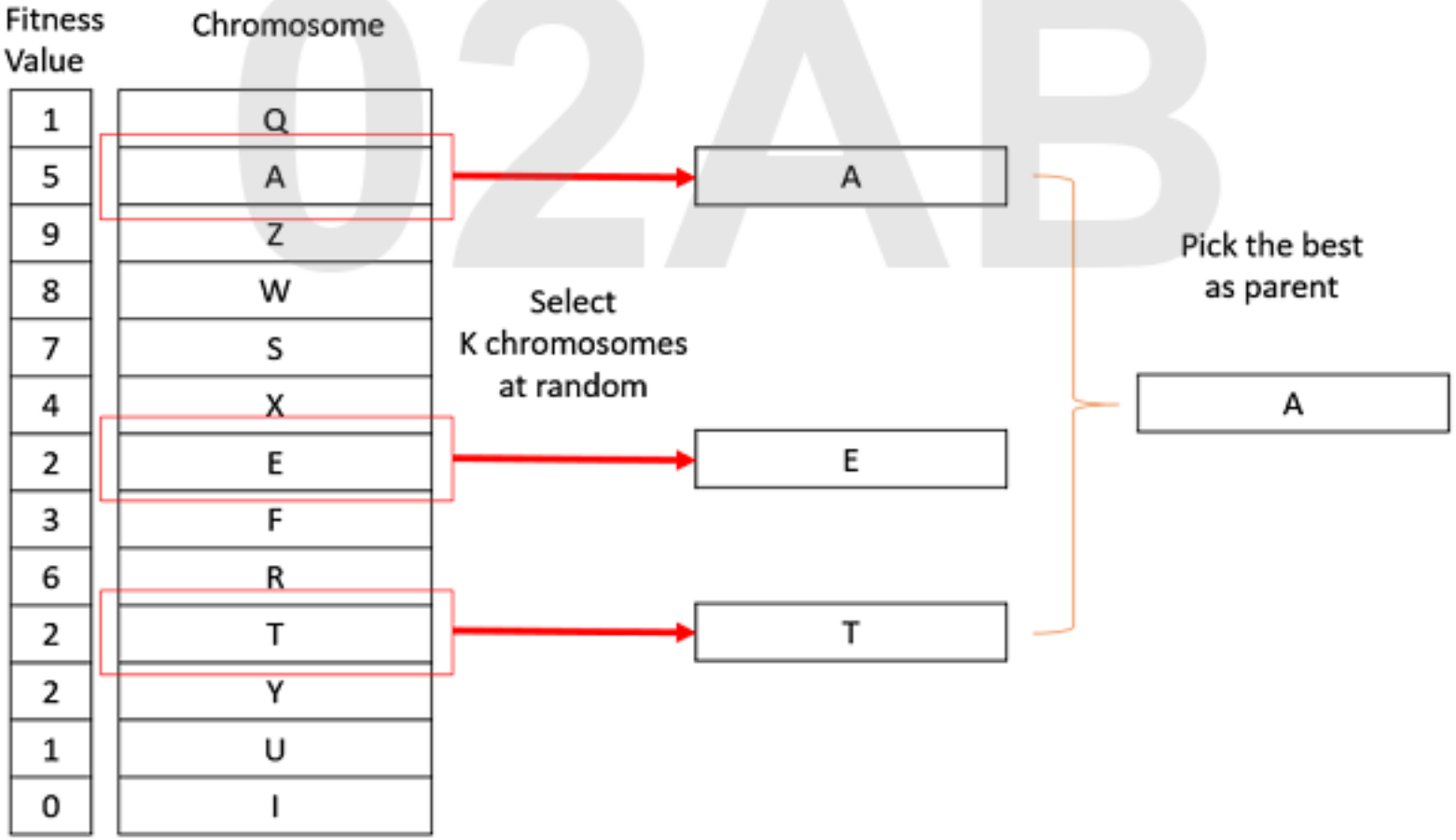
- Use multiple fixed points instead of only one fixed point.
- Parents are chosen in one spin → encourages the highly fit individuals to be chosen at least once.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

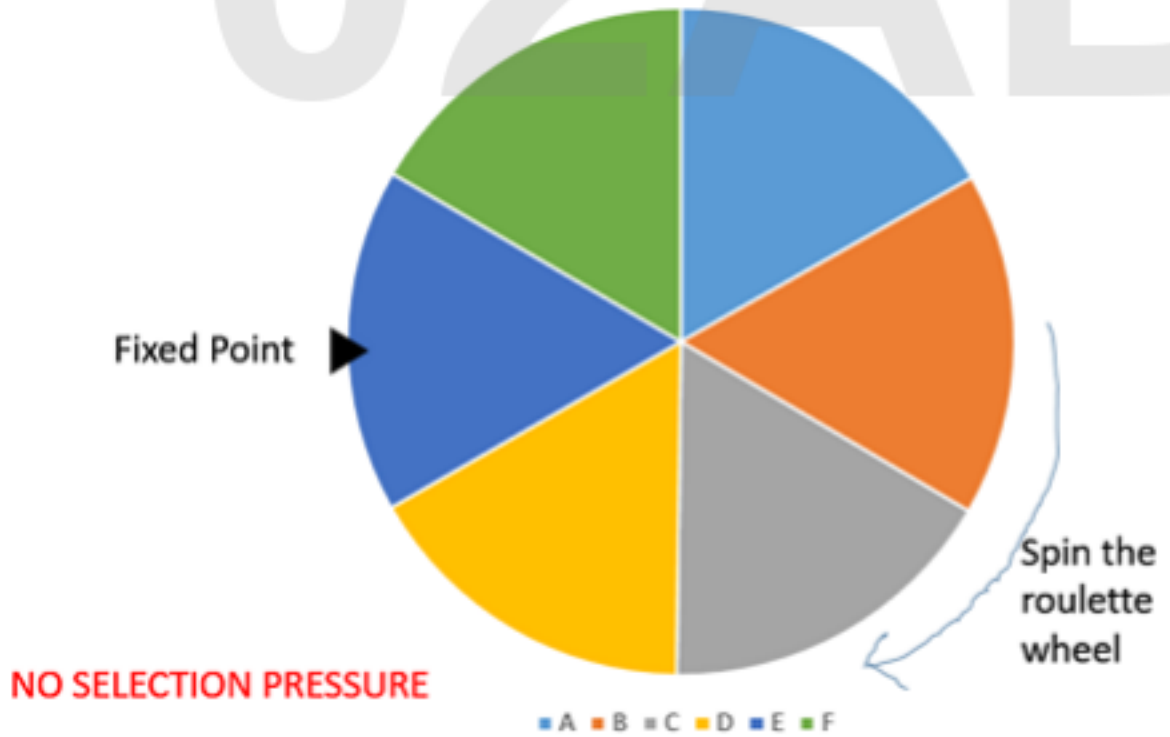
Tournament selection

- Select K individuals from the population at random, and
- Select the best out of these to become a parent.



Rank selection

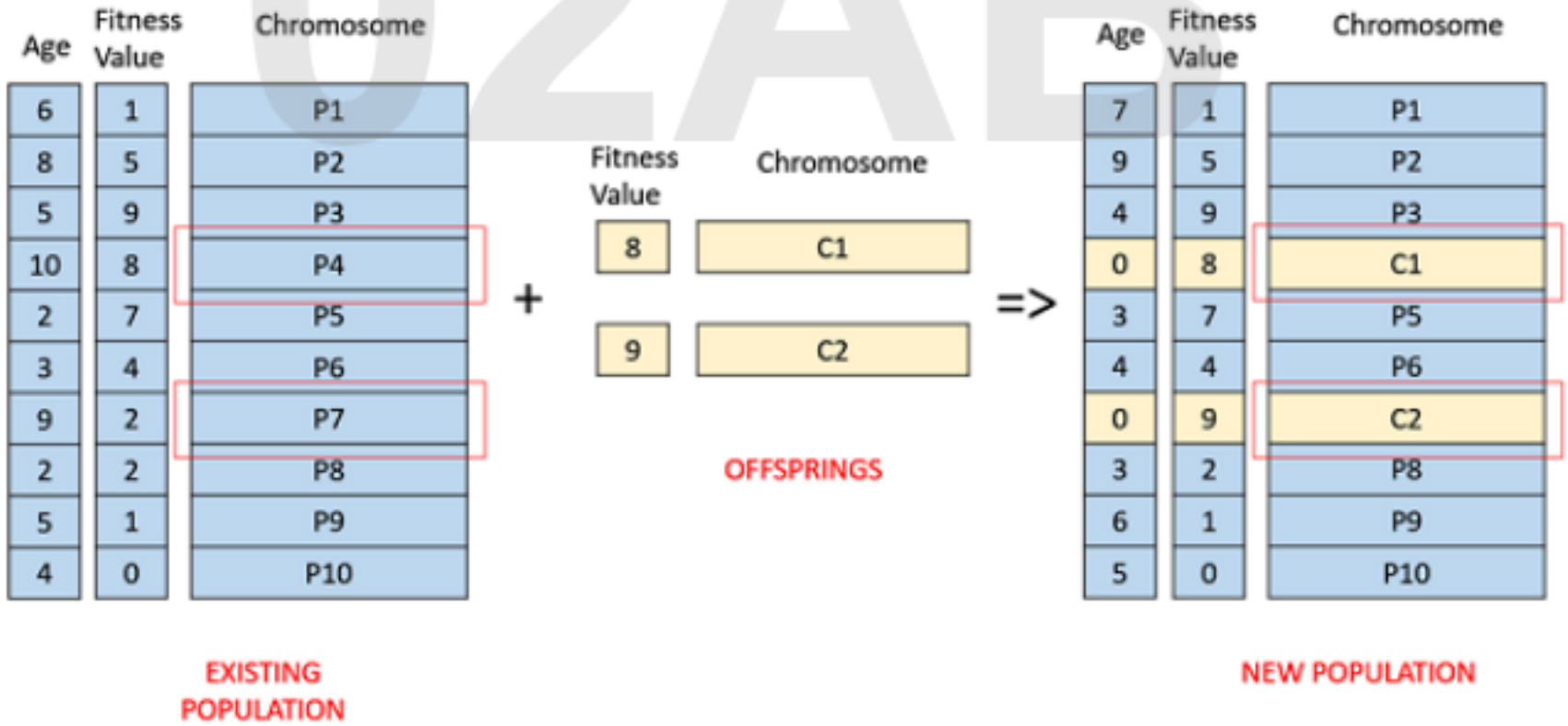
- How to do when individuals in the population have very close fitness values?
- Rank every individual in the population according to their fitness → selection based on ranks.



Chromosome	Fitness Value
A	8.1
B	8.0
C	8.05
D	7.95
E	8.02
F	7.99

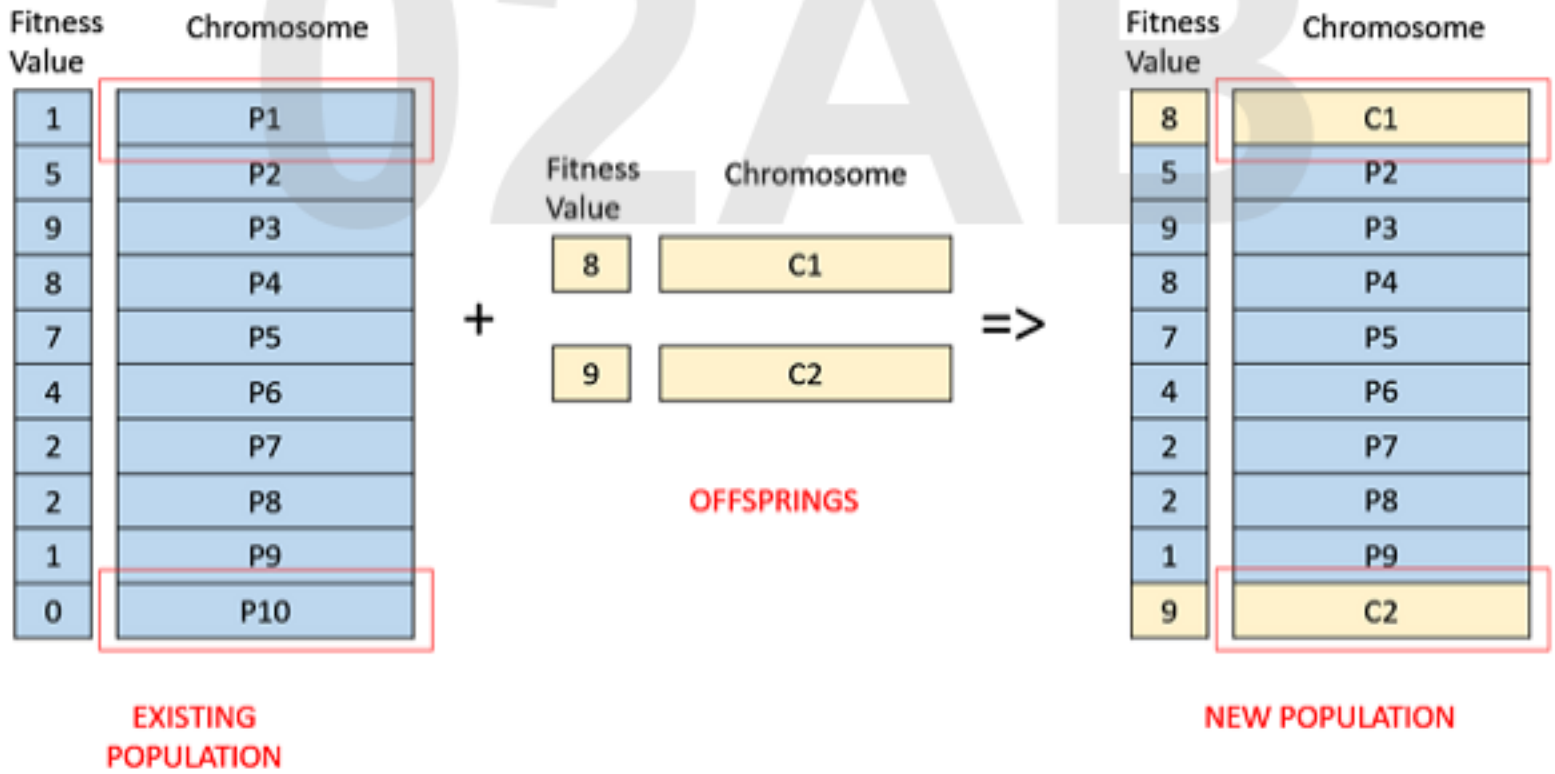
Age-based survivor selection

- Do not have a notion of a fitness function.
- Each individual is allowed in the population for a finite generation and then kicked out.



Fitness-based survivor selection

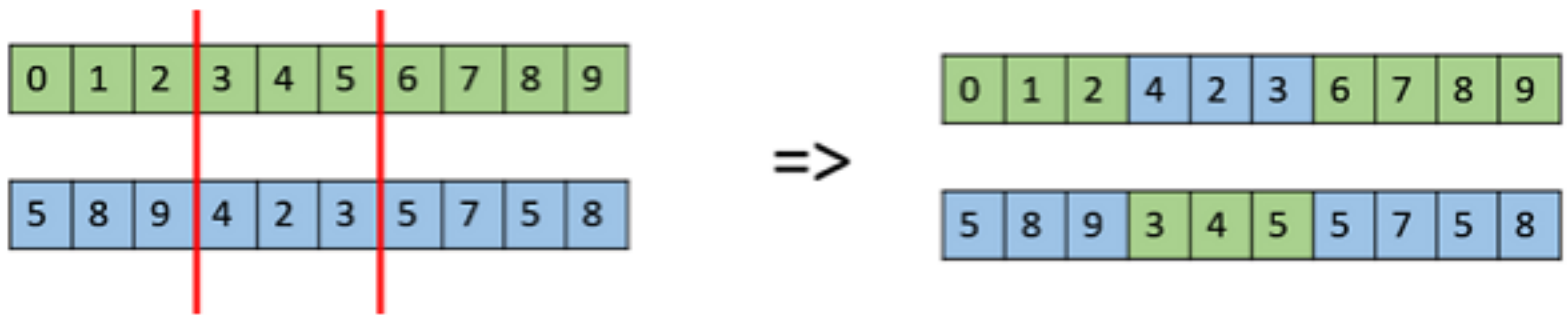
- The children replace the least fit individuals in the population.



Crossover

- Generate offspring(s) from 2 parents.
- Crossover types:
  - Single-point crossover

  - Two-point (multi-point) crossover

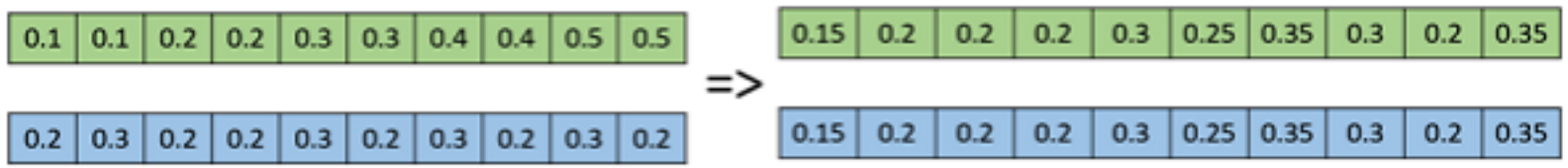


Crossover (cont.)

- Uniform crossover

- Whole arithmetic recombination

```
Child1 = α.x + (1-α).y
Child2 = α.x + (1-α).y, where x and y are parents
```



Crossover (cont.)

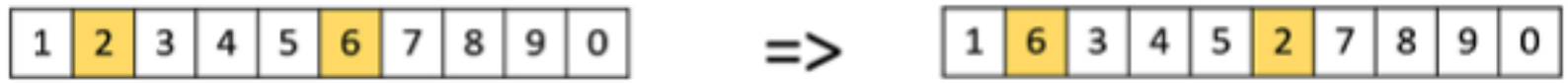
- Davis' order crossover (OX1):
  - Create two random crossover points in the parents and copy the middle segment of the 1st parent to the 1st offspring.
  - Copy the remaining unused numbers from the 2nd parent to the 1st child, wrapping around the list.
  - Repeat for the 2nd child with the parent's role reversed.

- Other crossovers: partially mapped crossover (PMX), order-based crossover (OX2), shuffle crossover, ring crossover, etc.

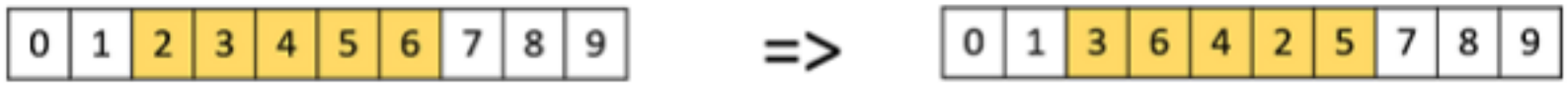
Mutation

- Generate new offspring from single parent
- Mutation types:
  - Bit flip mutation

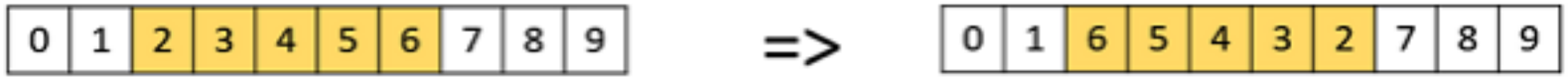
  - Random resetting: extension of the bit flip for the integer representation.
  - Swap mutation



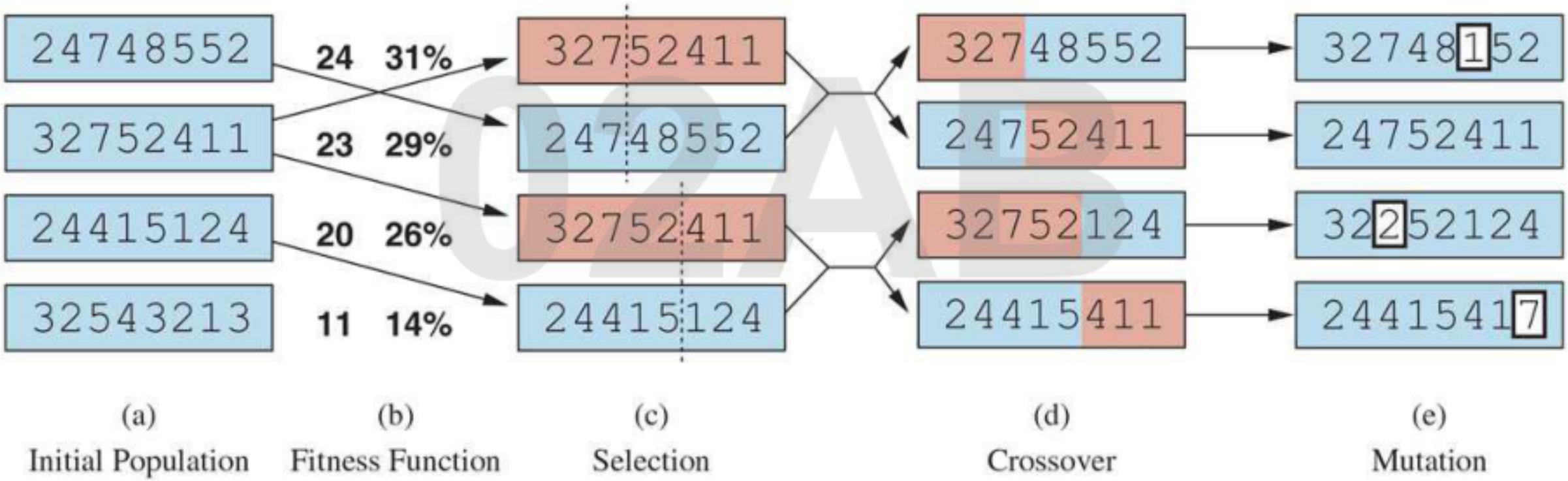
  - Scramble mutation



  - Inversion mutation

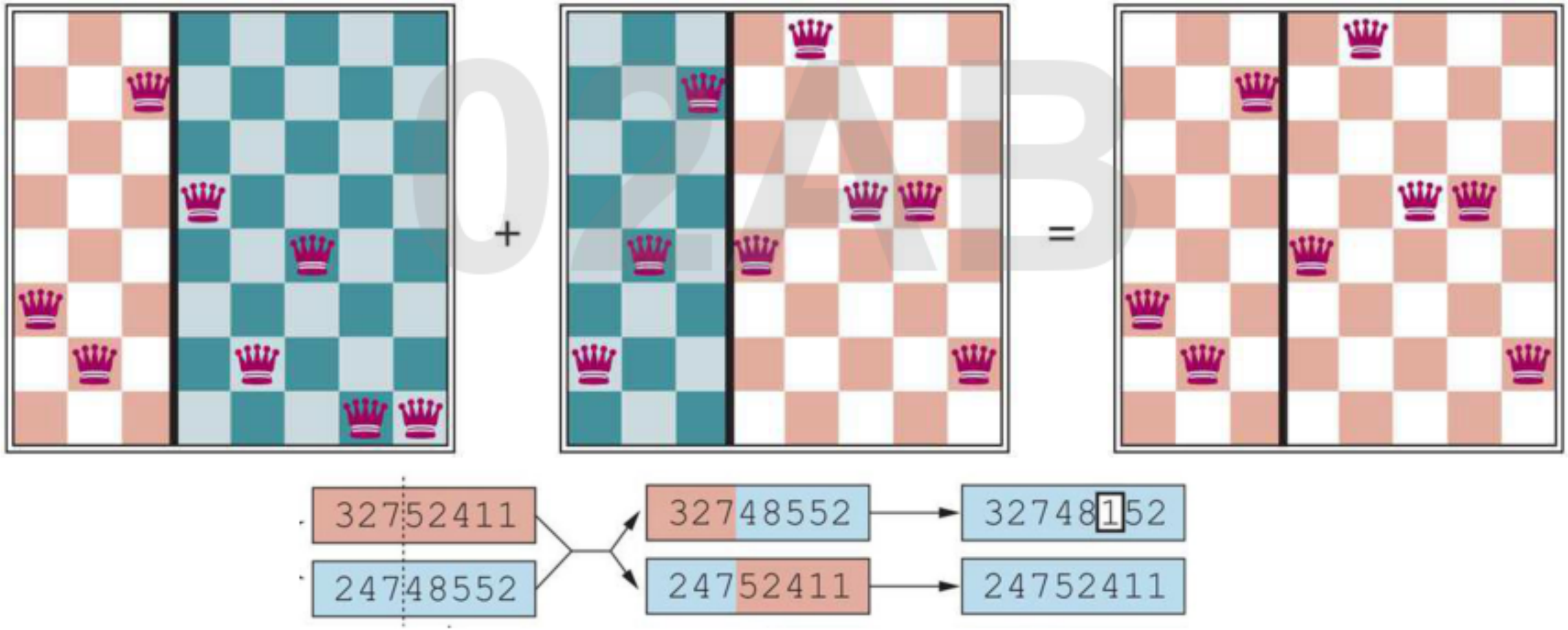


Example: Genetic algorithm



A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Example: Genetic algorithm



Genetic algorithm

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

No convergence rule or guarantee!

Example: Knapsack problem

	A	B	C	D
kg	7	2	1	9
$	5	4	7	2

Max Weight: 15kg

Example: Individual representation

	7	2	1	9
$	5	4	7	2
	A	B	C	D
	1	0	0	1
	A picked	B and C not picked		C picked

Example: Population initialization

0	1	0	0
0	1	0	1
1	1	1	1
0	0	1	0
1	0	0	0
1	0	1	0

kg	7	2	1	9
$	5	4	7	2
	A	B	C	D

Fitness coefficient

0	1	0	0	4
0	1	0	1	6
1	1	1	1	0
0	0	1	0	7
1	0	0	0	5
1	0	1	0	12

Example: Selection

Example: Crossover and mutation

- Crossover

- Mutation

Lecture summary

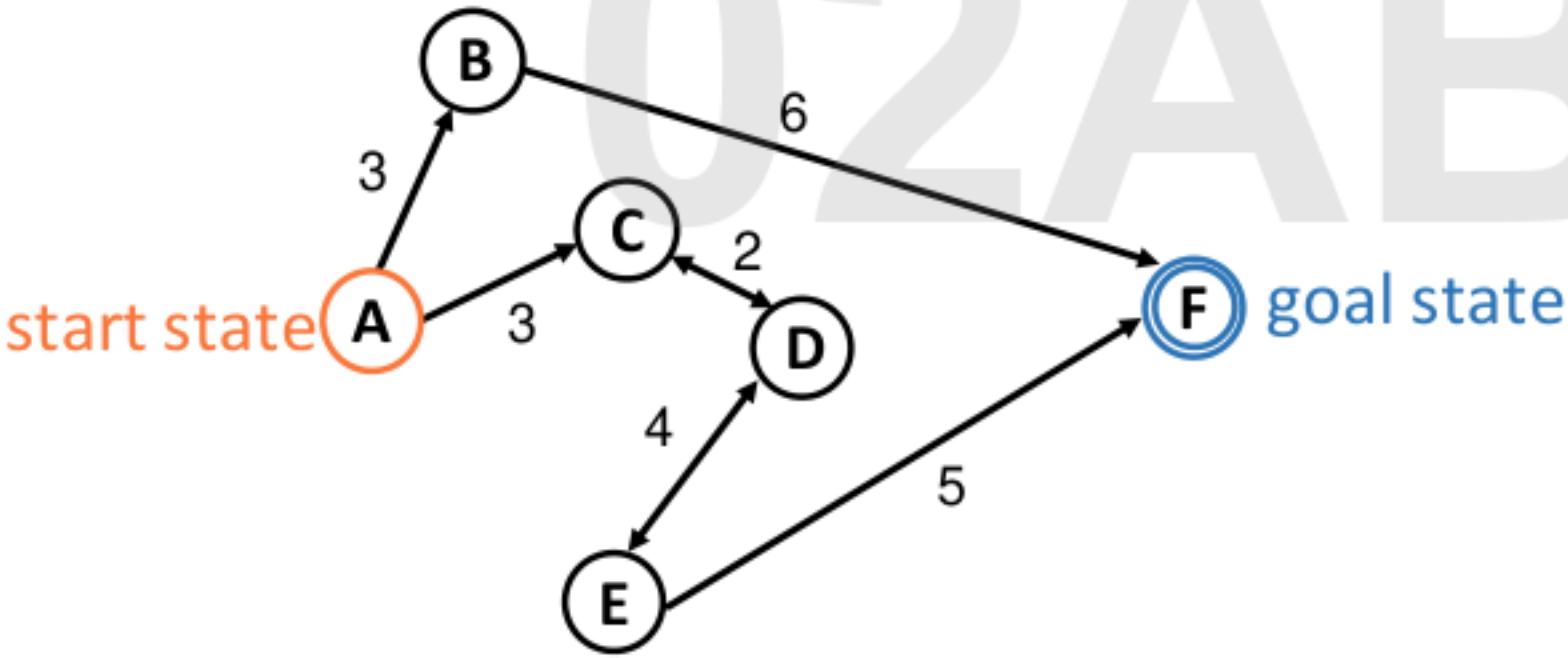
- Informed search algorithms vs. evaluation functions:

$$\begin{aligned}
\text{A* search:} &\quad g(n) + h(n) &\quad (W = 1)\\
\text{Uniform-cost search:} &\quad g(n) &\quad (W = 0)\\
\text{Greedy best-first search:} &\quad h(n) &\quad (W = \infty)\\
\text{Weighted A* search:} &\quad g(n) + W \times h(n) &\quad (1 < W < \infty)
\end{aligned}$$

- **Local search algorithms**: keep one or a few state(s) instead of all reached states.
- **Genetic algorithm**: maintain a population in which each individual is a possible solution. Evolute the population over the generations to create the best individual.

Exercises

- Show steps of the greedy first-search and A* search algorithms for the following route-finding problem?



h(n)	Est. value
h(A)	15
h(B)	14
h(C)	12
h(D)	10
h(E)	13
h(F)	0

Exercises

- Write Python functions for the following search algorithms
  - Greedy best-first search,
  - A* search,
  - Hill-climbing search.
- Show the steps of the three algorithms for Romania and 8-puzzle problems.
- Write Python code to demo the genetic algorithm for the Knapsack problem.