# Artificial Intelligence

Hai Thi Tuyet Nguyen

# Outline

# CHAPTER 4: INFORMED SEARCH AND EXPLORATION

4.1 Informed search algorithm

1. Greedy best-first search
2. A* search
3. Heuristics

4.2 Local search algorithms

1. Hill-climbing
2. Simulated annealing
3. Local beam search
4. Genetic algorithms

# 4.1 Informed search algorithm

# 4.1 Informed search algorithm

Informed search strategy uses problem-specific knowledge beyond the definition of the problem itself => can find solutions more efficiently than can an uninformed strategy.

1. Greedy Best-first search
2. A* search

# 4.1 Review: Graph search

**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  ***initialize the explored set to be empty***
  **loop do**
    **if** the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    **if** the node contains a goal state **then return** the corresponding solution
    ***add the node to the explored set***
    expand the chosen node, adding the resulting nodes to the frontier
      ***only if not in the frontier or explored set***

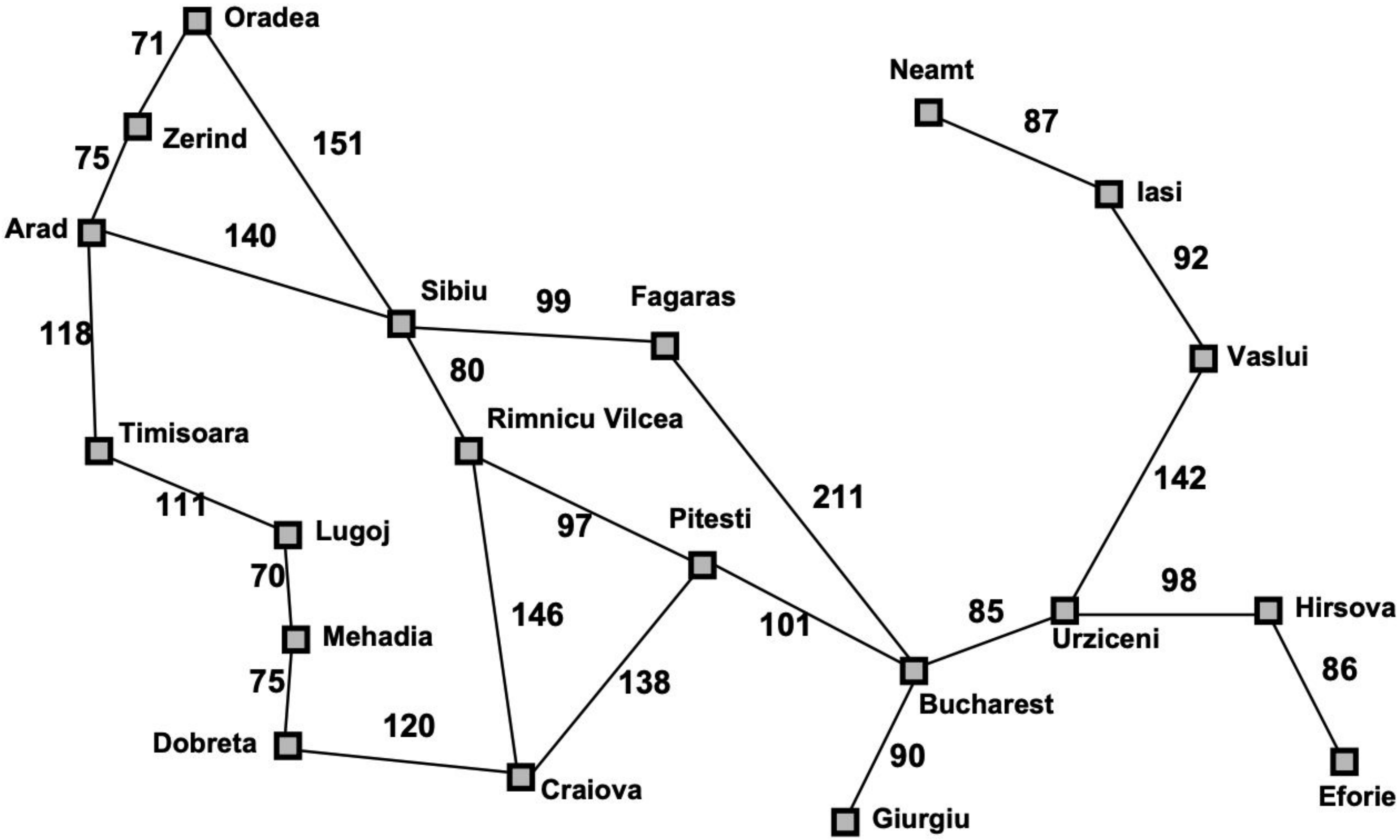
# 4.1 Best-first search

- **Best-first search:**
  - is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm
  - a node is selected for expansion based on ***an evaluation function***, ***f(n)***.
    - ***f(n)*** = a cost estimate function => the node with the lowest evaluation is expanded first.
  - Most best-first algorithms include a heuristic function, ***h(n)***:
    - ***h(n)*** = estimated cost of the cheapest path from the state at node *n* to a *goal state*.

- **Implementation:**
  - similar to UCS with frontier as a queue sorted in ***order of f(n)***

- **Special cases:** The choice of ***f(n)*** determines the search strategy
  - greedy best first search
  - A* search

# 4.1 Romania with step costs 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 |

# 4.1.1 Greedy best first search

- Greedy best first search: expands the node that appears to be closest to goal
- Evaluation function **f(n)**:

  $\mathbf{f(n) = h(n)}$

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

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

CH(*problem*) **returns** a solution, or failure

*problem*.INITIAL-STATE, PATH-COST = 0
lered by PATH-COST, with *node* as the only element

**rst search**

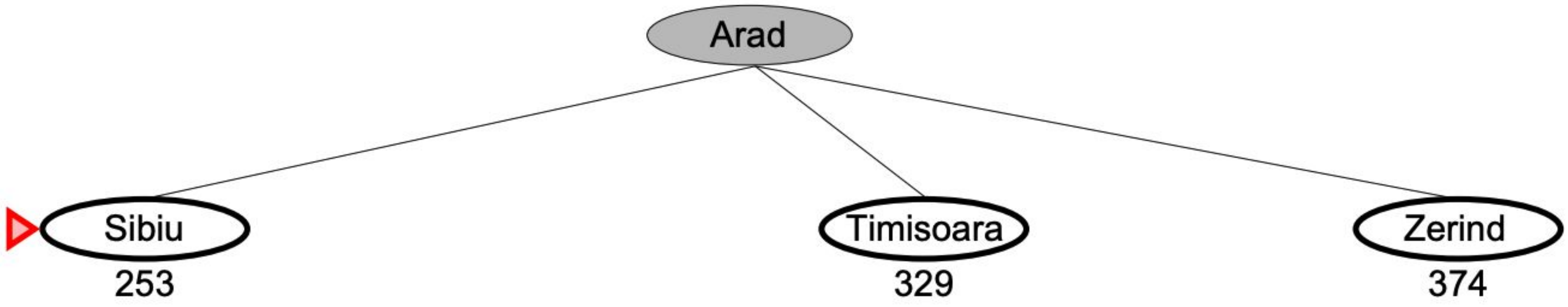

**return** failure
chooses the lowest-cost node in *frontier* */
*de*.STATE) **then return** SOLUTION(*node*)
*d*
*n*.ACTIONS(*node*.STATE) **do**
(*problem*, *node*, *action*)
*explored* or *frontier* **then**
(*child*, *frontier*)
n *frontier* with higher PATH-COST **then**
r node with *child*

node: (A, 366)
frontier: (A, 366)
explored:

Loop 1:
node: (A, 366)
frontier:
explored: (A, 366)
For
child:

| | | | |
|---|---|---|---|
| **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 |

# 4.1.1 Greedy best first search




| | | | |
|---|---|---|---|
| **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 |

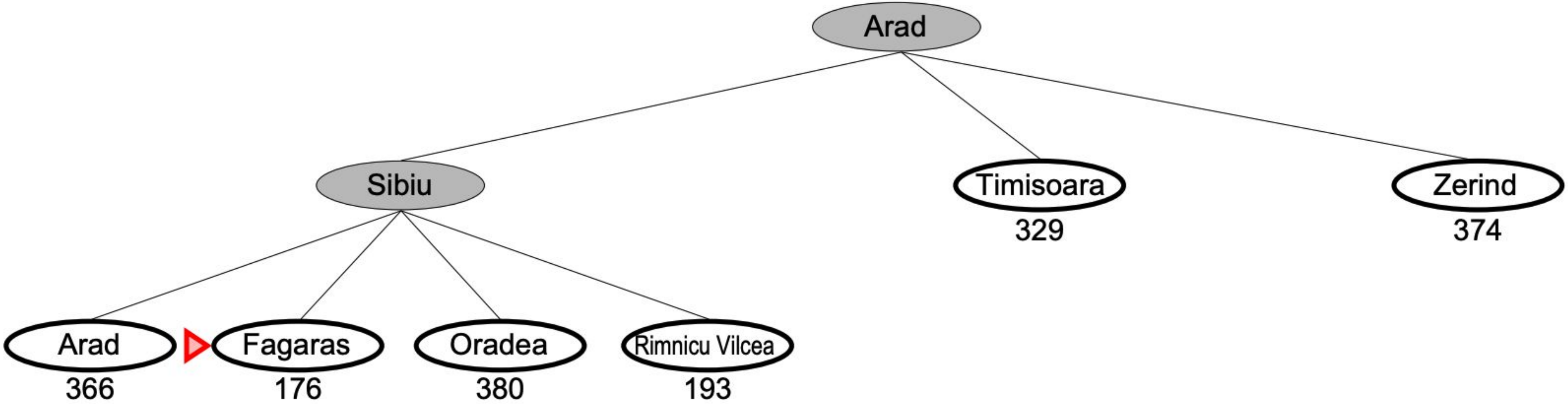
# 4.1.1 Greedy best first search

Loop 1:
node: (A, 366)
frontier:
explored: (A, 366)
    for
        child: (S,253)
        frontier: (S,253)
        explored: (A, 366)

        child: (T,329)
        frontier: (S,253), (T,329)
        explored: (A, 366)

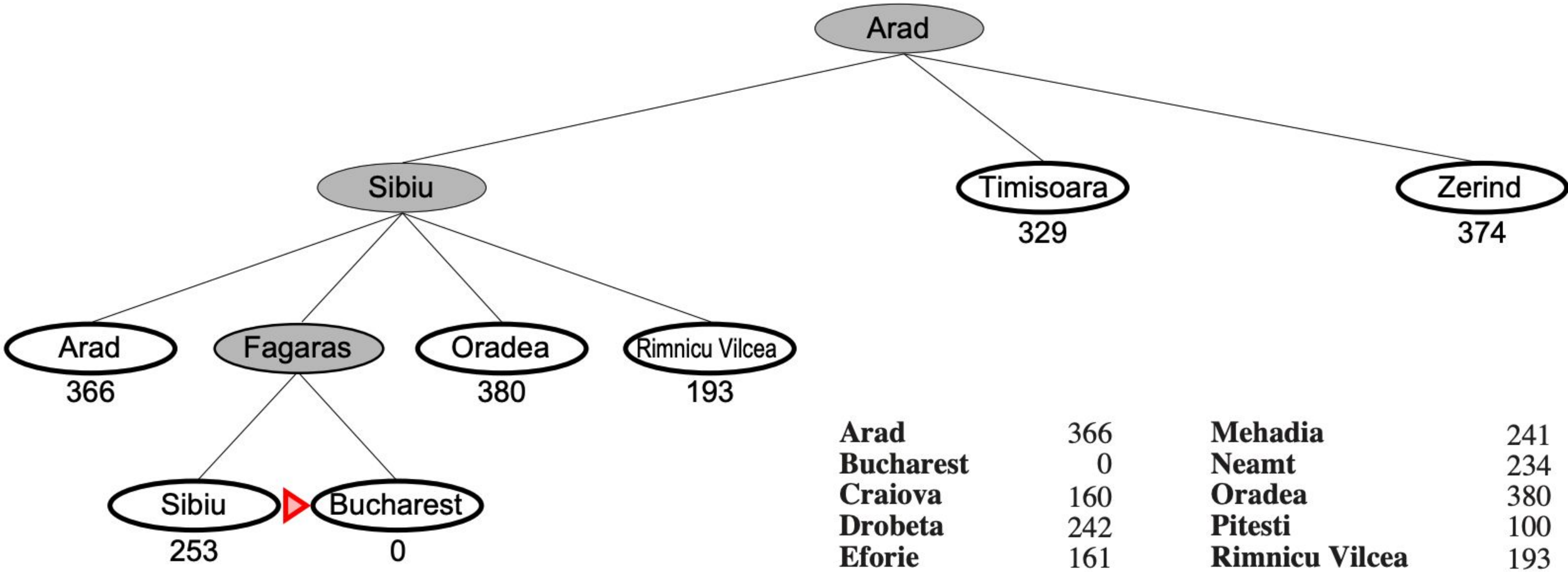
        child: (Z,374)
        frontier: (S,253), (T,329), (Z,374)
        explored: (A, 366)

# 4.1.1 Greedy best first search




| | | | |
|---|---|---|---|
| **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 |

# 4.1.1 Greedy best first search



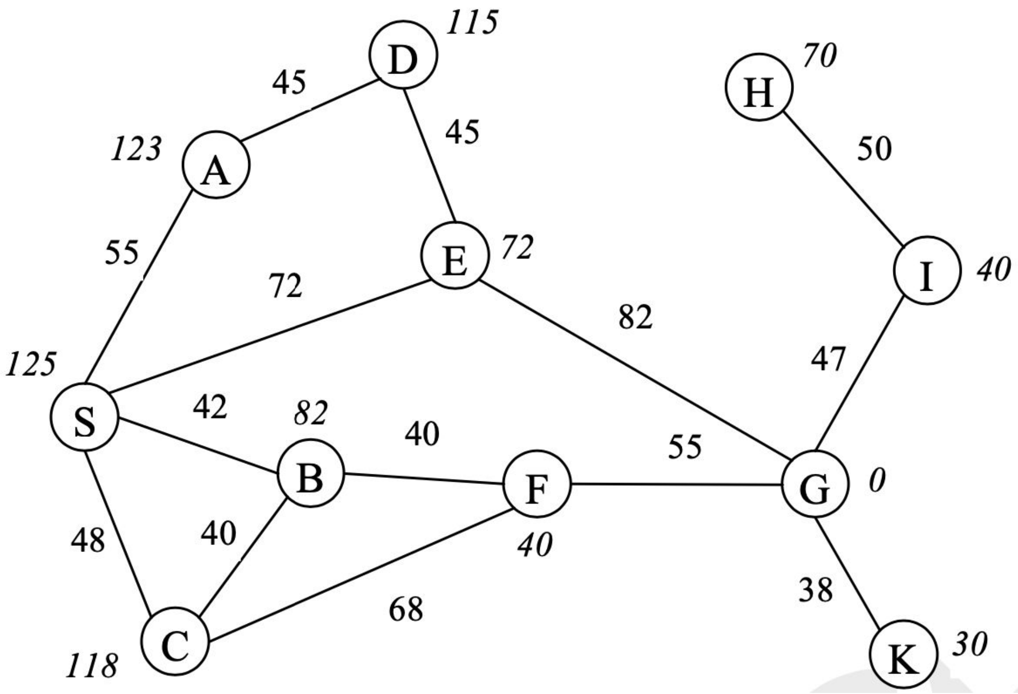

| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara**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

# 4.1.1 Properties of greedy best first search

- Complete:
  - Tree search: no, can get stuck in loops
  - Graph search: complete in finite space
- Time: $O(b^m)$, but a good heuristic can give dramatic improvement
- Space: $O(b^m)$, keeps all nodes in memory
- Optimal: No

# 4.1.1 Properties of greedy best first search

- Greedy search found path 1 with path cost as 154: S -> E -> G
- Optimal path with path cost as 137: S -> B -> F -> G



# 4.1.2 A* search

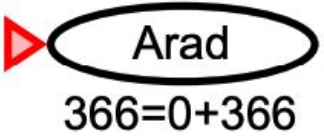
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$

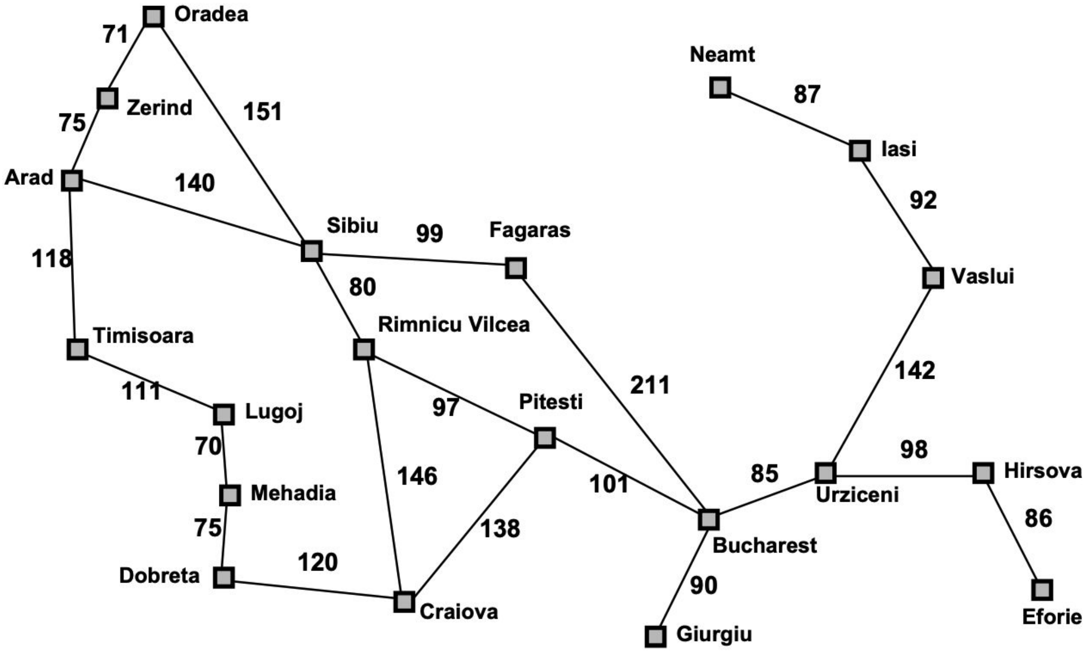
  $g(n)$ = cost from initial state to $n$

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

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

# 4.1.2 A* search example

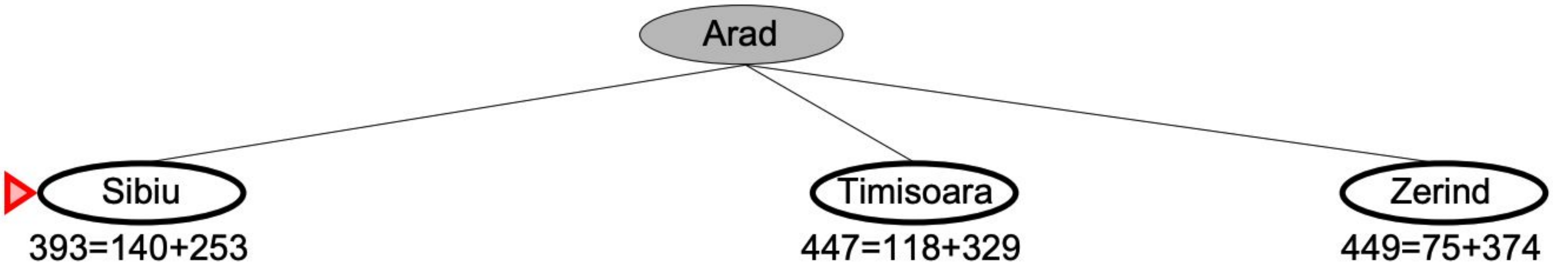


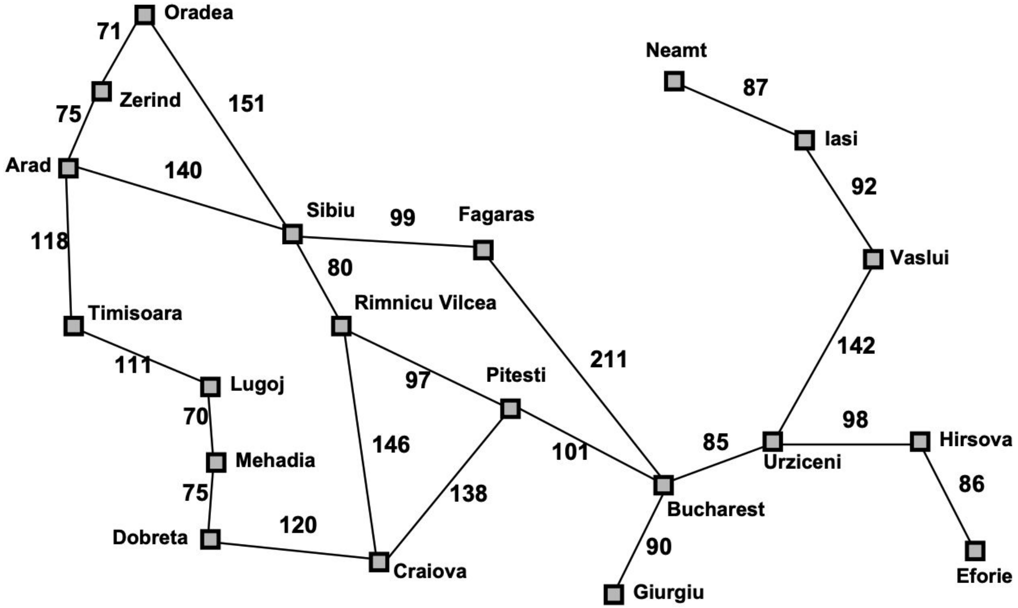




Straight−line distance
to Bucharest

| | |
|---|---|
| **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 |

# 4.1.2 A* search example

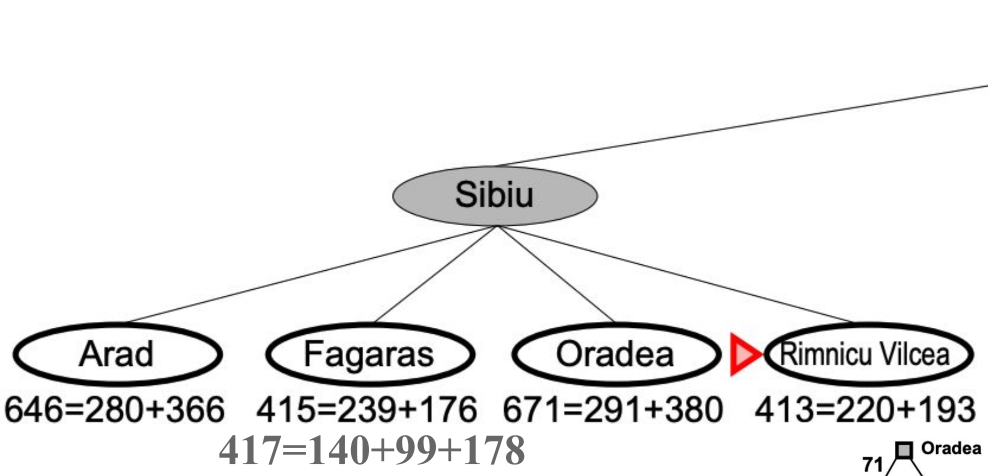


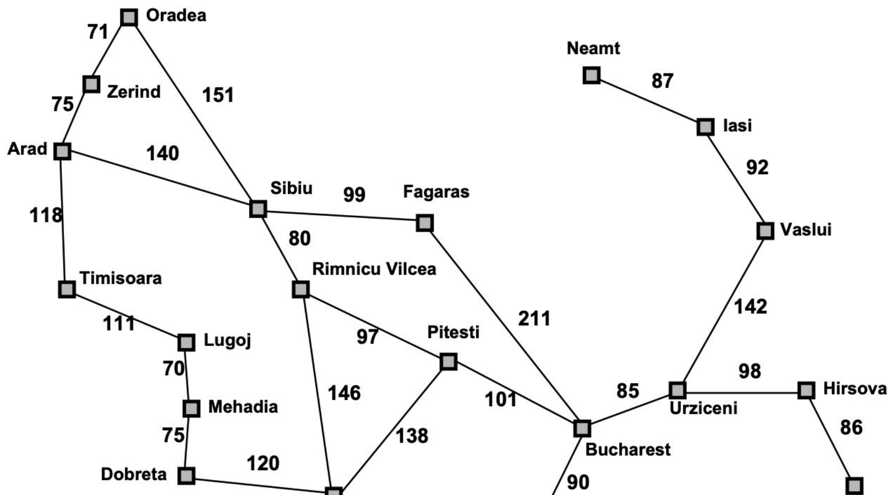




Straight−line distance
to Bucharest

| | |
|---|---|
| **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 |

# 4.1.2 A* search example

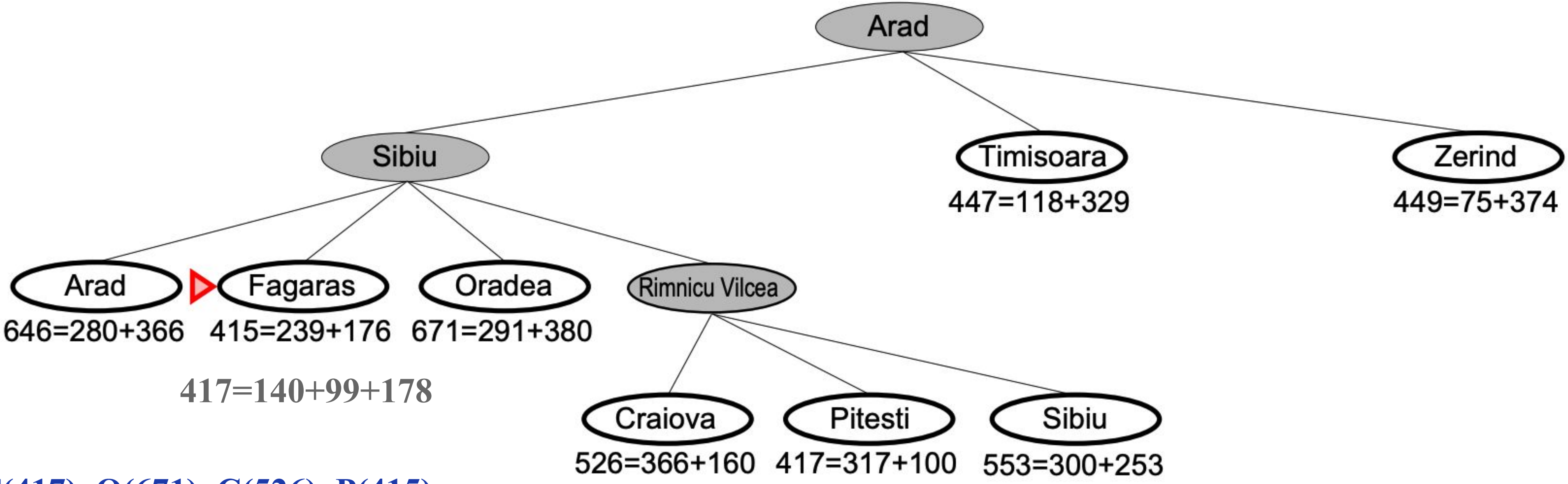



Straight−line distance to Bucharest

| | |
|---|---|
| **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 |

# 4.1.2 A* search example



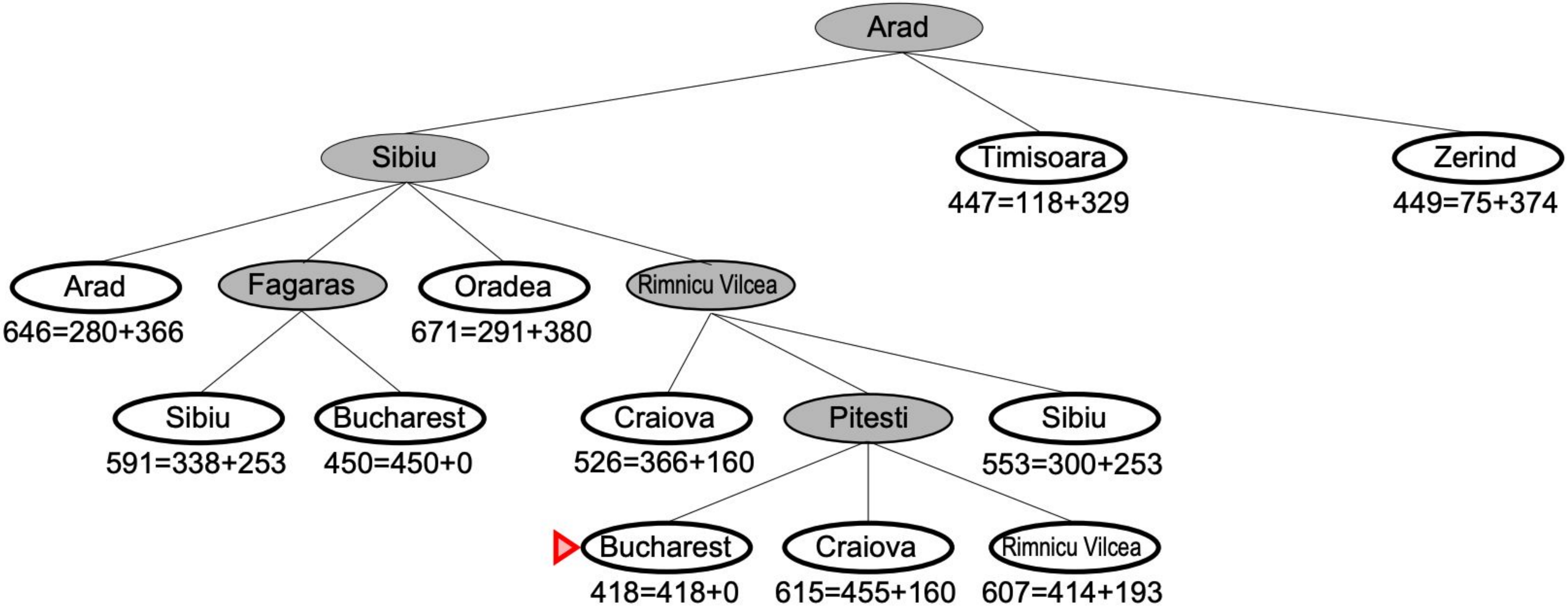

(417), O(671), C(526), P(415)

*415=140+80+97+98*


Straight−line distance to Bucharest

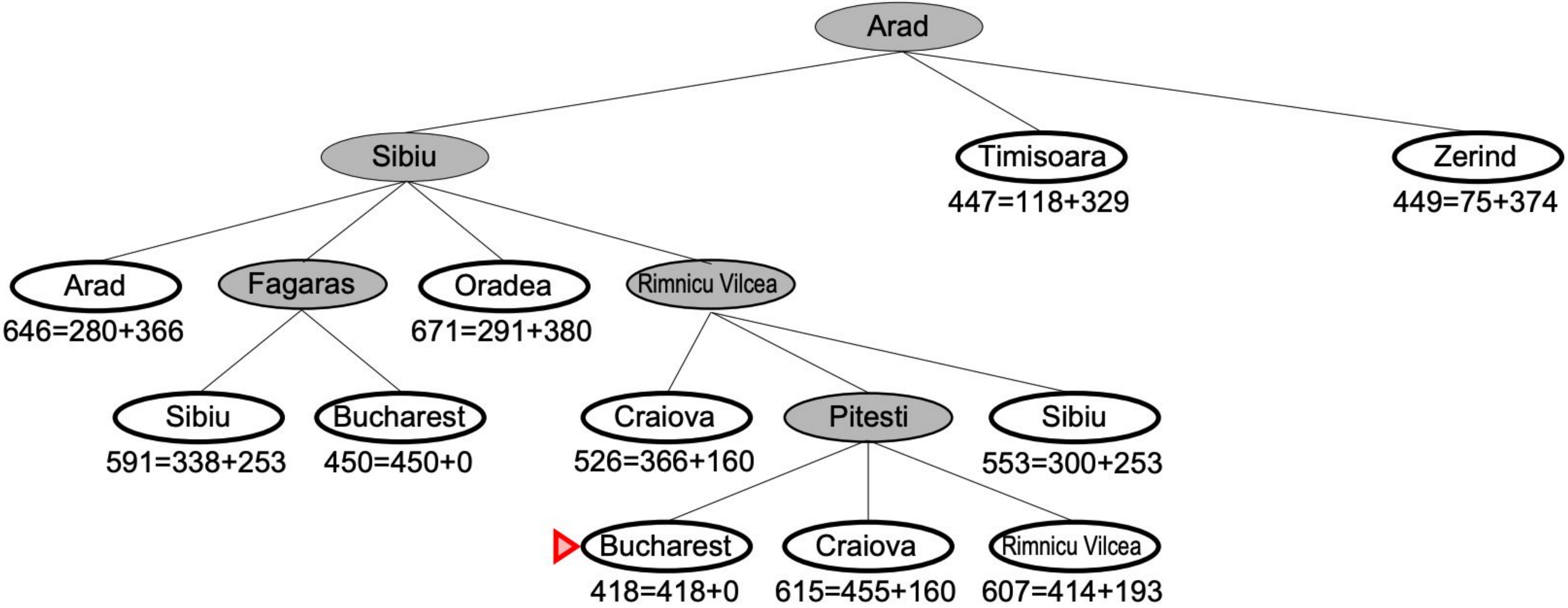
| | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |

# 4.1.2 A∗ search example



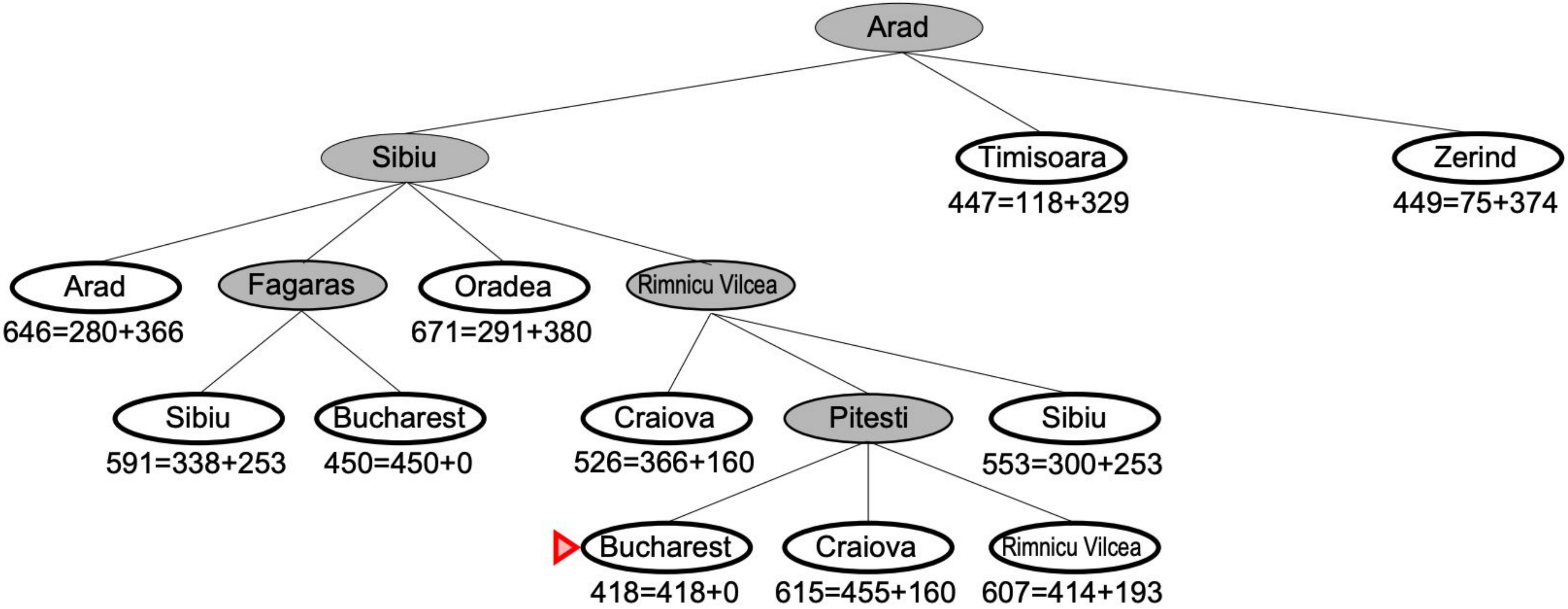

**Frontier: T(447), Z(449), <u>F(417)</u>, O(671), C(526), *B(418)***

# 4.1.2 A* search example




**Frontier: T(447), Z(449), O(671), C(526), B(418)**

# 4.1.2 A∗ search example

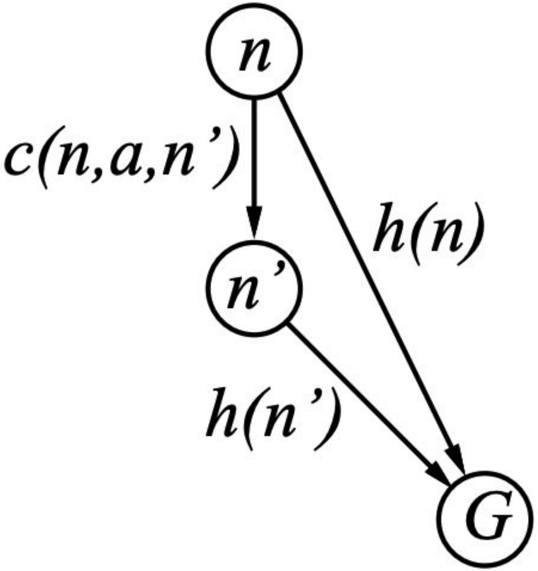



**Frontier: T(447), Z(449), O(671), C(526), <u>B(418)</u>**

# 4.1.2 Properties of A*

- Complete: yes, if all step costs > ε and if b is finite
- Time: $O(b^m)$
- Space: $O(b^m)$, keeps all nodes in memory
- Optimal:
  - tree-search: optimal if ***h(n)*** is ***admissible***
    - ***h(n)*** never overestimates the cost to reach the goal
  - graph-search: optimal if ***h(n)*** is ***consistent***
    - ***n' = RESULT(n, a)***

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

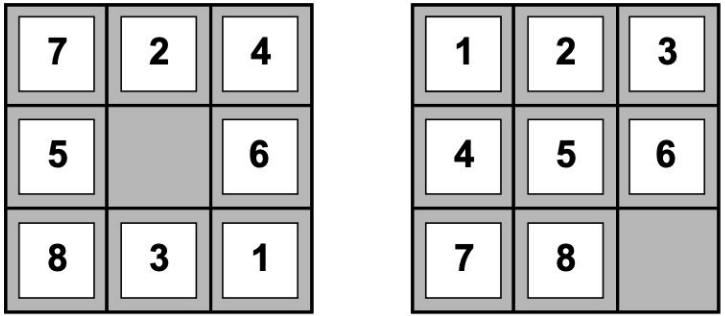


# 4.1.3 Heuristic functions

- The performance of heuristic search algorithms depends on the quality of the heuristic function.

- Good heuristics can sometimes be constructed:
  - relaxing the problem definition
  - precomputing solution costs for subproblems in a pattern database
  - learning from experience with the problem class

# 4.1.3 Heuristic functions




Start State    Goal State

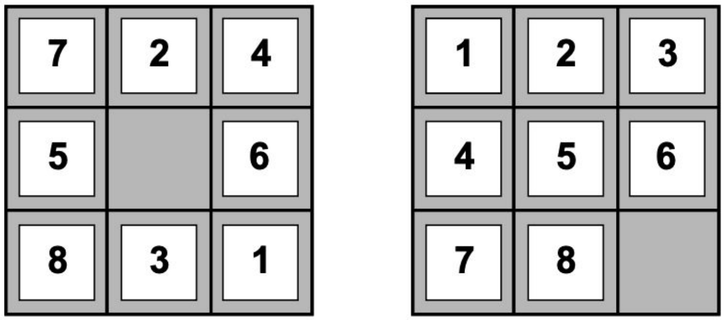
Look at two *admissible* heuristics for the 8-puzzle:

- $h1(n)$ = the number of misplaced tiles.
- $h2(n)$ = the sum of the distances of the tiles from their goal positions, i.e., Manhattan distances

$h1(S) = 6$

$h2(S) = 4+0+3+3+1+0+2+1 = 14$

# 4.1.3 Heuristic functions




Start State    Goal State

Look at two *admissible* heuristics for the 8-puzzle:

- $h1(n)$ = the number of misplaced tiles.
- $h2(n)$ = the sum of the distances of the tiles from their goal positions, i.e., Manhattan distances

$h1(S) = 6$

$h2(S) = 4+0+3+3+1+0+2+1 = 14$

If $h2(n) \geq h1(n)$ for all $n$ then $h2$ ***dominates*** $h1$ and is better for search

Given any admissible heuristics $h_a$, $h_b$,

$$h(n) = max(h_a(n), h_b(n))$$

is also admissible and dominates $h_a$, $h_b$

# 4.1.3 Heuristic functions

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

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

**Figure 3.29** Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with $h_1$, $h_2$. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths $d$.

# 4.2 Local search algorithms

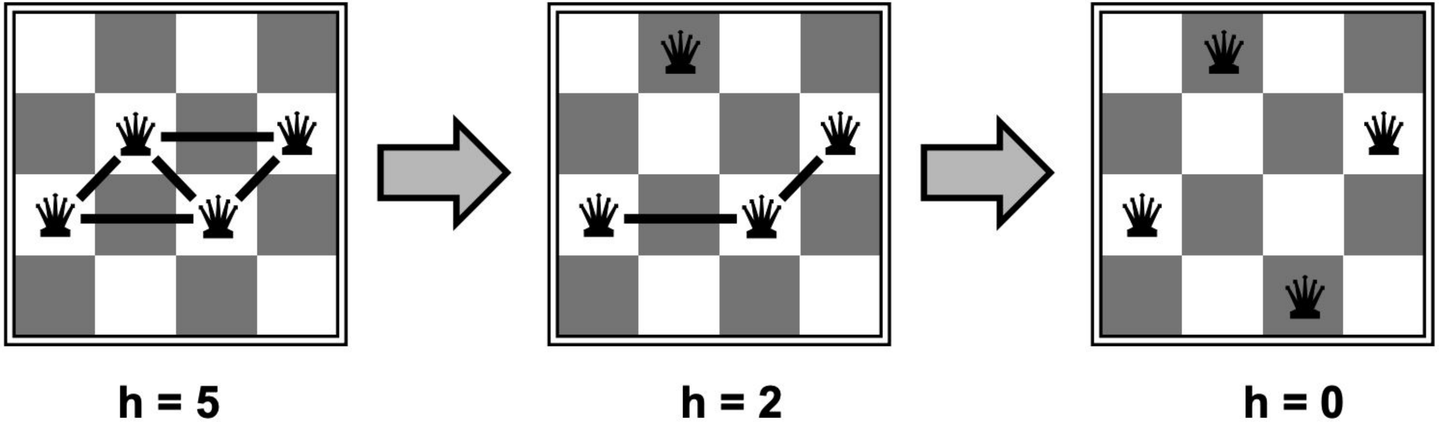
# 4.2 Local search algorithms

- In many optimization problems, the solution is the goal state, not the path
  - find the configuration satisfying constraints, e.g., timetable, n-queen
  - find the optimal configuration, e.g., Travelling Salesperson Problem (TSP)
- In these cases, we can use the local search algorithms:

  keep a single “current state”, try to improve it

# 4.2 Example: n-queens

- 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 a number of conflicts



# 4.2 Local search algorithms

State space landscape:

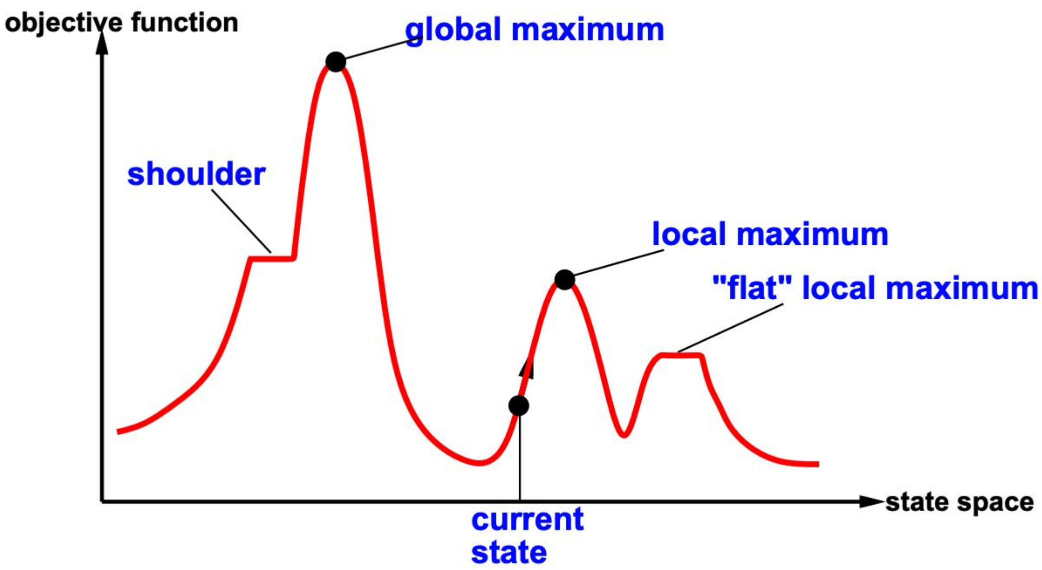
- "location": state
- "elevation": the value of objective function; find the highest peak - a global maximum

The aim is to find the global maximum.

Local search modifies the current state to try to improve it, as shown by the arrow.

Local search algorithms

1. Hill-climbing
2. Annealing Simulated
3. Genetic algorithm



# 4.2 Hill-Climbing

- Hill climbing known as greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.
- How it works:
  - continually moves in the direction of increasing value, i.e., uphill.
  - terminates when it reaches a "peak" where no neighbor has a higher value

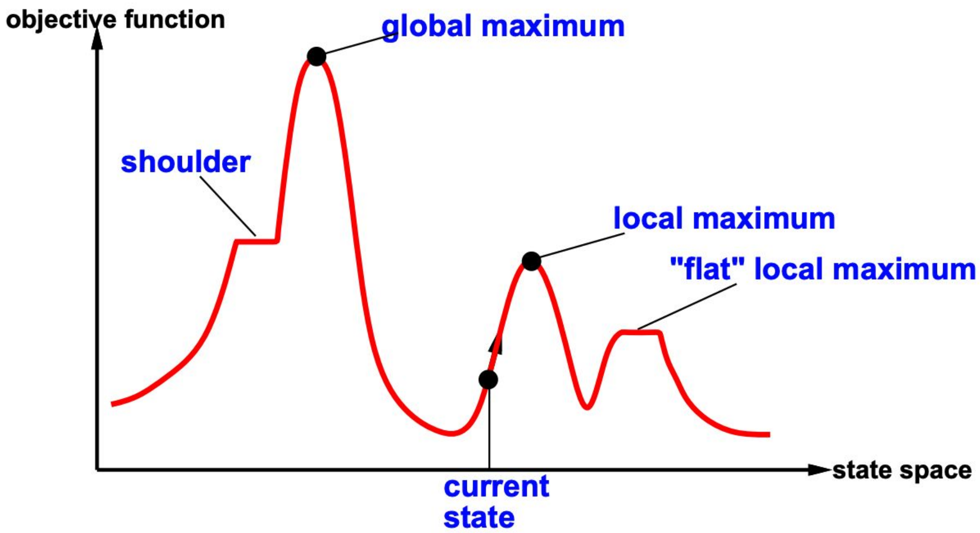
**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

    *current* ← MAKE-NODE(*problem*.INITIAL-STATE)
    **loop do**
        *neighbor* ← a highest-valued successor of *current*
        **if** neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
        *current* ← *neighbor*

# 4.2 Hill-Climbing

**Hill climbing often gets stuck for the following reasons:**

- Local maximum: a peak that is higher than each of its neighboring states, but lower than the global maximum.
- Ridges: they result in a sequence of local maximum
  => very difficult for greedy algorithms to navigate.
- Plateaux: an area of the state space landscape where the evaluation function is flat



# 4.2 Hill-Climbing

**Variants of hill-climbing:**

- Stochastic hill climbing: chooses at random from among the uphill moves
- First-choice hill climbing: generates successors randomly until one is generated that is better than the current state.
- Random-restart hill climbing:
  - it conducts a series of hill-climbing searches from randomly generated initial state, stopping when a goal is found
  - The success of hill climbing depends on the shape of the state-space landscape: if it has few local maxima and plateaux -> random-restart hill climbing will find a good solution very quickly

# 4.2 Simulated annealing search

- Idea: escape local maxima by allowing some bad moves, but gradually decrease their size and frequency
  - Simulated annealing search picks a random move.
  - If the move improves the situation, it is accepted.
  - Otherwise, the algorithm accepts the move with some probability.
    - The probability decreases with the “badness” of the move, $\Delta E$ by which the evaluation is worsened.
    - The probability decreases as the “temperature” T goes down

# 4.2 Simulated annealing search

```
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 prob. 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^(Δ E/T)
```

# 4.2 Local beam search

- Idea: keeps track of $k$ states rather than just one
  - It begins with $k$ randomly generated states.
  - At each step, all the successors of all $k$ states are generated.
  - If any one is a goal, the algorithm halts.
  - Otherwise, it selects the $k$ best successors from the complete list and repeats.

- Problem: quite often, all k states end up on same local hill
- Idea: choose $k$ successors randomly, biased towards good ones ~ stochastic beam search

# 4.2 Genetic algorithms

- A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states

# 4.2 Genetic algorithms (GAs)

(a) GAs begin with a set of ***k*** randomly generated ***states***, called the ***population***

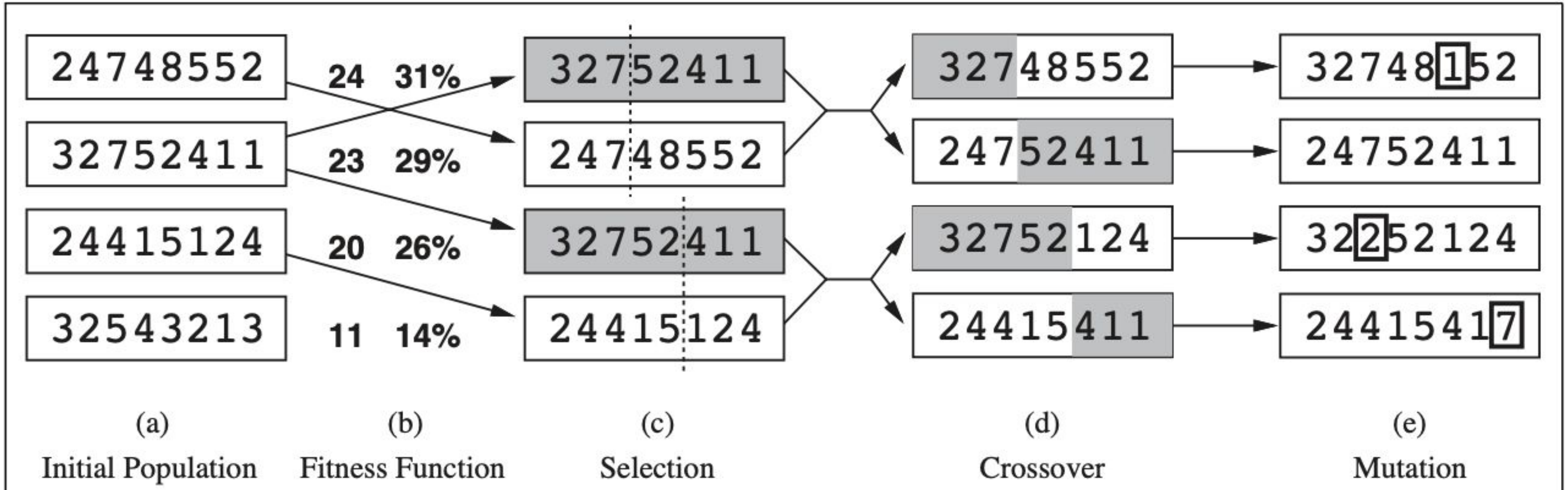



**Figure 4.6** The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

# 4.2 Genetic algorithms (GAs)

(b) Each state is rated by the **objective function**, i.e., the **fitness function**.

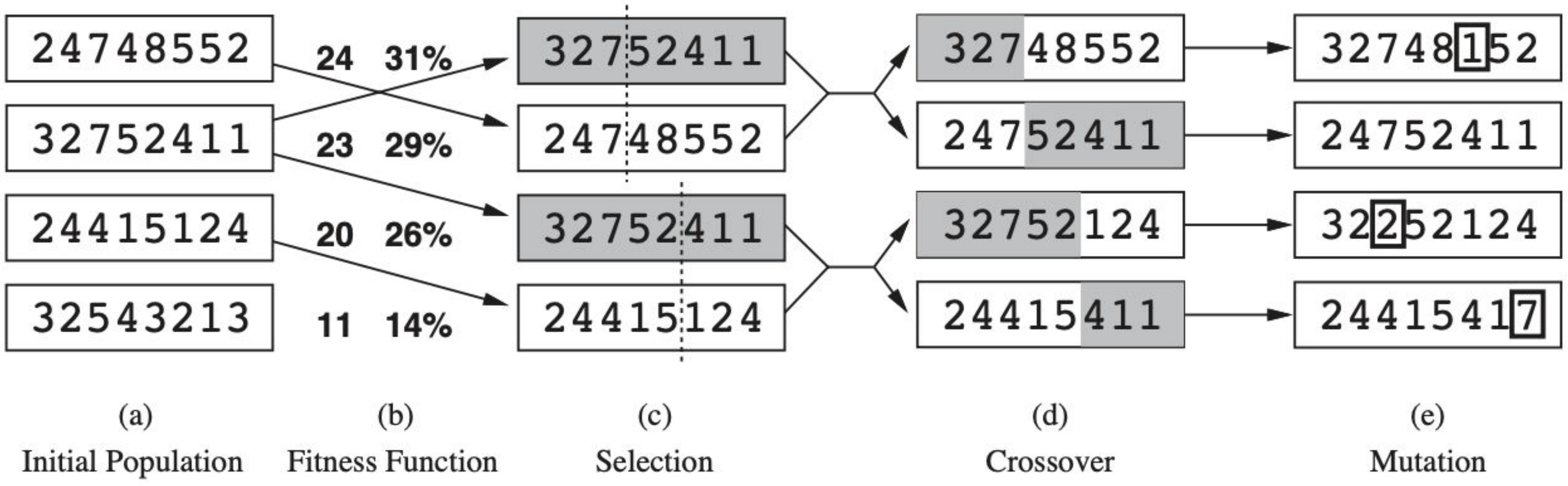



**Figure 4.6** The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

# 4.2 Genetic algorithms (GAs)

(c) 2 pairs are selected at random for ***reproduction***, in accordance with the probabilities in (b)
Each pair to be mated, a ***crossover point*** is chosen randomly from the positions in the string.

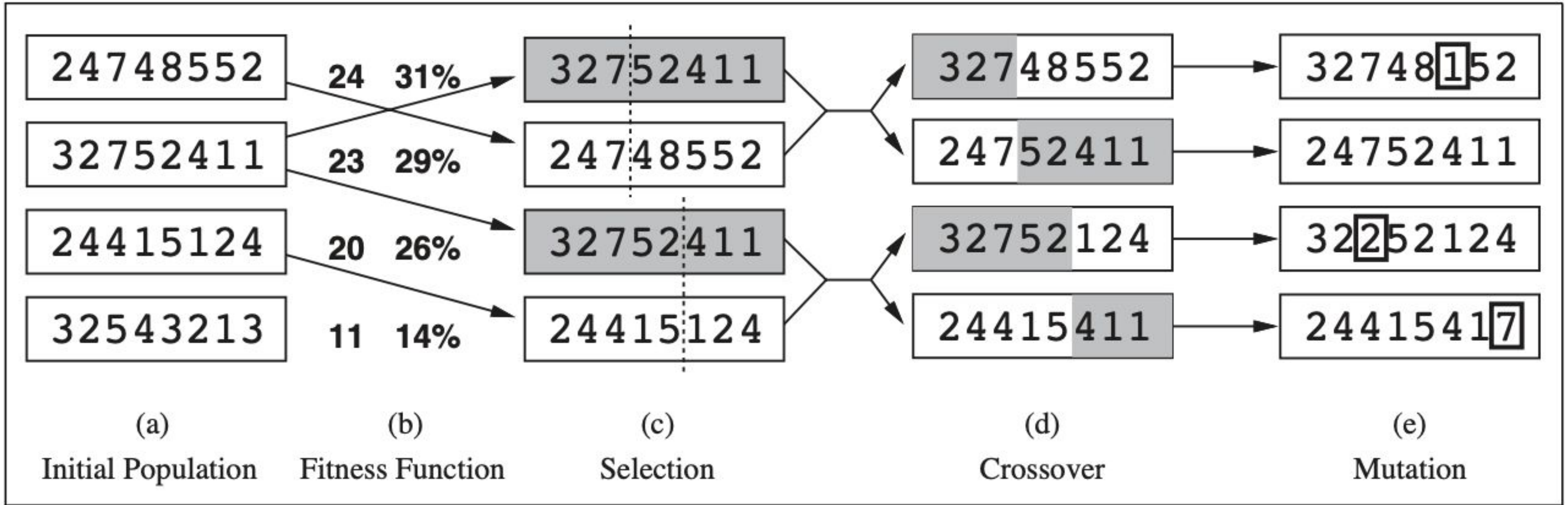



**Figure 4.6** The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

# 4.2 Genetic algorithms (GAs)

(d) the offspring themselves are created by ***crossing over*** the parent strings at the crossover point

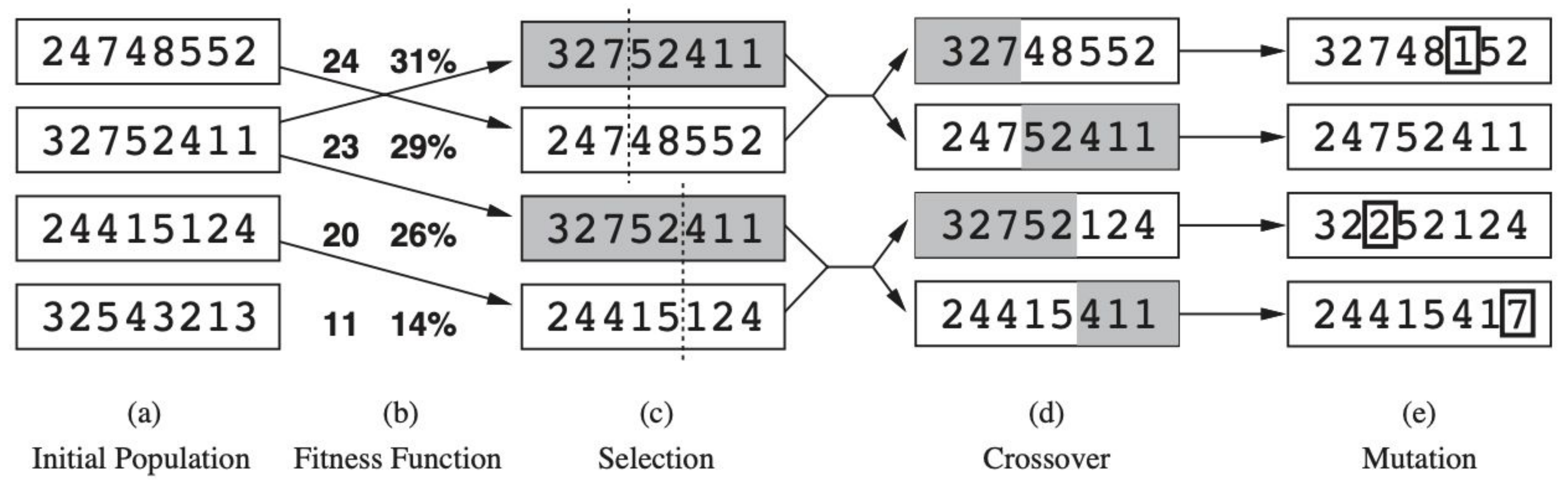



**Figure 4.6** The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

# 4.2 Genetic algorithms (GAs)

(e) each location is subject to ***random mutation*** with a small independent probability.

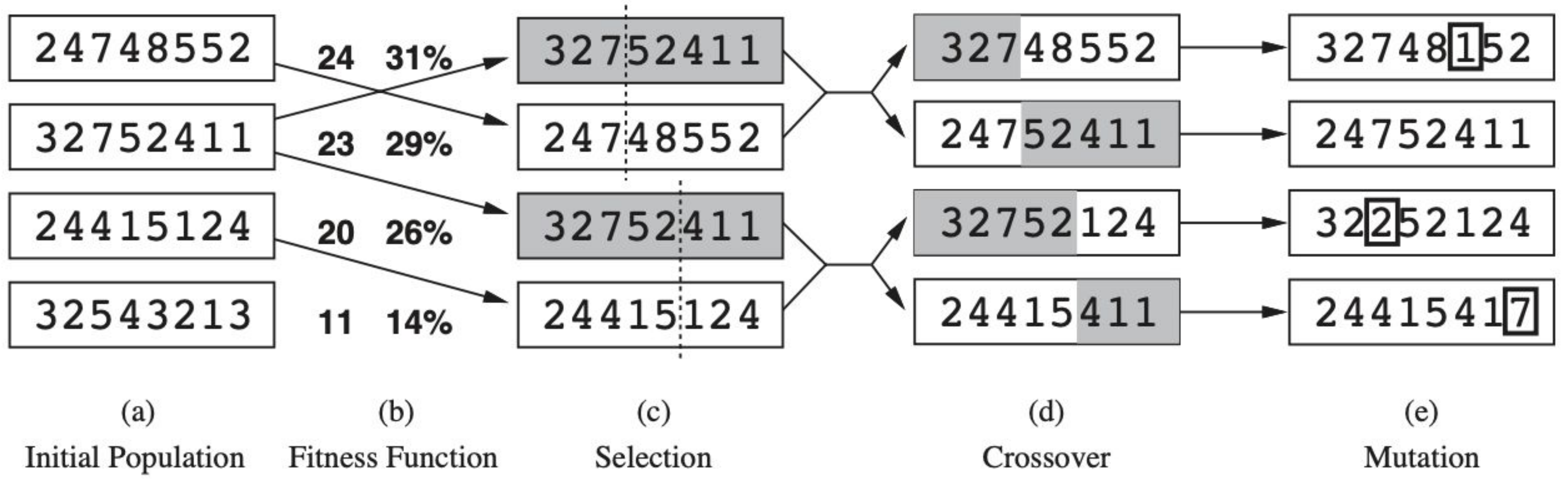



**Figure 4.6** The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

# 4.2 Genetic algorithms (GAs)

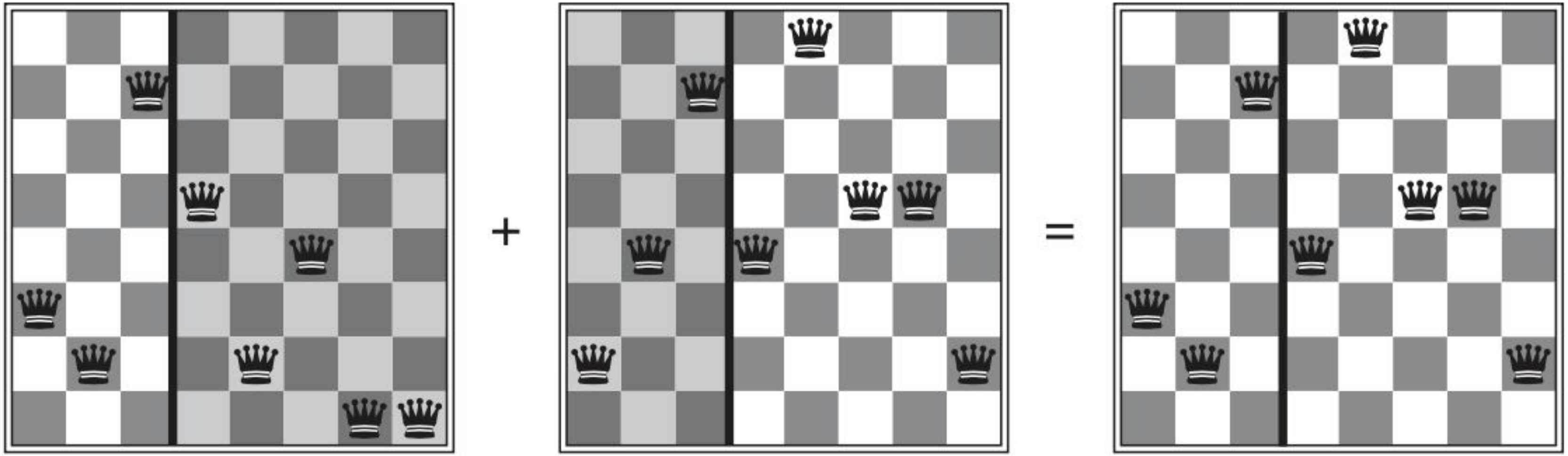



**Figure 4.7** The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

# 4.2 Genetic algorithms (GAs)

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

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

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

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

**Figure 4.8** A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.